

5

(19) 日本国特許庁 (J P)

(12) 公開特許公報 (A)

(11) 特許出願公開番号  
特開2002-116916  
(P2002-116916A)

(43) 公開日 平成14年4月19日 (2002. 4. 19)

(51) Int.Cl. <sup>7</sup>	識別記号	F I	テマコード <sup>*</sup> (参考)
G 0 6 F 9/45		G 0 6 F 9/32	3 4 0 B 5 B 0 1 3
9/32	3 4 0	9/38	3 7 0 X 5 B 0 3 3
9/38	3 7 0	9/44	3 2 2 F 5 B 0 8 1

審査請求 有 請求項の数15 O L (全 30 頁)

(21) 出願番号 特願2000-304618(P2000-304618)

(22) 出願日 平成12年10月4日 (2000. 10. 4)

(71) 出願人 390009531

インターナショナル・ビジネス・マシー  
ズ・コーポレーション  
INTERNATIONAL BUSIN  
ESS MACHINES CORPO  
RATION  
アメリカ合衆国10504、ニューヨーク州  
アーモンク (番地なし)

(74) 復代理人 100104880

弁理士 古部 次郎 (外4名)

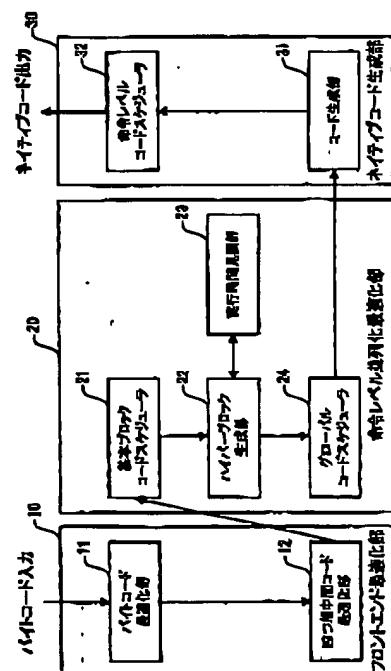
最終頁に続く

(54) 【発明の名称】 プログラムの最適化方法及びこれを用いたコンパイラ

(57) 【要約】

【課題】 高速かつ適切に、プログラムをハイパーブ  
ロックに分割することにより、プログラムの実行効率を向  
上させることを可能とする。

【解決手段】 処理対象のプログラムに対し、基本ブ  
ロックを単位として実行時間を見積もる基本ブロック・コ  
ードスケジューラ21と、基本ブロックを並列実行領域  
であるハイパーブロックにまとめるハイパーブロック生  
成部22及び実行時間見積部23とを備えたコンパイラ  
であり、実行時間見積部23は、基本ブロックの接続が  
条件分岐を伴う場合に、前記基本ブロック単位で見積も  
られた実行時間に基づいて、条件分岐のまま実行する場  
合と、条件分岐部分を並列実行する場合の実行時間を見  
積もり、ハイパーブロック生成部22は、並列実行する  
方が実行時間が短い場合、この部分を一まとまりの並列  
実行領域とし、条件分岐のまま実行する方が実行時間が  
短い場合、この部分をこの条件分岐にて接続された複数  
の並列実行領域に分割する。



(2)

特開2002-116916

## 【特許請求の範囲】

【請求項1】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もるステップと、

前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成するステップと、

前記ネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もるステップと、

前記見積もりにより、並列実行する方が実行時間が短い場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短い場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割するステップとを含むことを特徴とするプログラムの最適化方法。

【請求項2】 前記基本ブロックを単位として実行時間を見積もるステップは、

前記基本ブロックを単位として見積もられた実行時間に基づいて、さらに、前記基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを取得するステップを含む請求項1に記載のプログラムの最適化方法。

【請求項3】 前記ネスト木を生成するステップは、前記基本ブロック間の依存関係を表す依存グラフを生成するステップと、

前記依存グラフから冗長な枝を取り除いた先行制約グラフを生成するステップと、

前記先行制約グラフのノードの接続関係を入れ子構造で表現することにより前記ネスト木を生成するステップとを含む請求項1に記載のプログラムの最適化方法。

【請求項4】 前記条件分岐部分の実行時間を判断するステップは、

前記子ノードにおいて実行可能な並列度ごとに、当該子ノードを並列実行した場合の実行時間の最大値を求めるステップと、

各並列度における前記実行時間の最大値のうち特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含む請求項1に記載のプログラムの最適化方法。

【請求項5】 前記条件分岐部分の実行時間を判断するステップは、

実行時間の判断に先立って、前記子ノードを構成する前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正するステップを含む請求項1に記載のプログラムの最適化方

法。

【請求項6】 前記プログラムの並列実行領域を決定するステップは、

前記複数の子ノードを複数の並列実行領域に分割する場合に、ハードウェアが持つ並列度で各子ノードを並列実行した場合における当該各子ノードの実行時間を比較するステップと、

前記実行時間が最も短い子ノードを残して、他の子ノードを独立した並列実行領域とするステップとを含む請求項1に記載のプログラムの最適化方法。

【請求項7】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムの条件分岐部分を、プレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能な計算機にて並列実行する場合の実行時間を見積もるステップと、

見積もられた前記実行時間が、前記条件分岐部分をそのまま実行した場合の実行時間よりも短い場合に、当該条件分岐部分を前記プレディケート付き命令による並列実行を行うように書き換えるステップとを含み、

前記並列実行時の実行時間を見積もるステップは、前記プログラムに対し、基本ブロックを単位として実行時間を見積もり、当該実行時間に基づいて、各基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを求めるステップと、

前記条件分岐による分岐先である基本ブロックを、前記クリティカルパス長及び平均並列度の情報に基づいて、実行可能な並列度ごとに、当該基本ブロックを並列実行した場合の実行時間の最大値を求めるステップと、各並列度における前記実行時間の最大値のうちの特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含むことを特徴とするプログラムの最適化方法。

【請求項8】 前記基本ブロックのクリティカルパス長及び平均並列度とを求めるステップは、前記基本ブロックを、隣り合う2辺の一方の値を前記クリティカルパス長とし、他方の値を前記平均並列度値とすると共に、クリティカルパス長に対応する辺が当該クリティカルパス長を下回らない範囲で変形可能とした矩形で表現するステップを含む請求項7に記載のプログラムの最適化方法。

【請求項9】 前記条件分岐部分の実行時間を見積もるステップは、実行時間の判断に先立って、前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正するステップを含む請求項7に記載のプログラムの最適化方法。

【請求項10】 プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの

(3)

特開2002-116916

最適化を行うコンパイラにおいて、  
処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もる第1のコードスケジューラと、  
前記基本ブロックをまとめて並列実行領域であるハイパーブロックを生成するハイパーブロック生成部と、  
前記プログラムにおける所定の領域を実行した場合の処理時間を見積もることにより、前記ハイパーブロック生成部による前記ハイパーブロックの生成を支援する実行時間見積部と、  
生成された前記ハイパーブロックごとにコードスケジューリングを行う第2のコードスケジューラとを備え、  
前記実行時間見積部は、  
前記基本ブロックの接続関係を入れ子構造で表すネスト木の所定のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もり、  
前記ハイパーブロック生成部は、  
前記実行時間見積部の見積もりにより、並列実行する方が実行時間が短いノードに関して、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いノードに関して、当該ノードの複数の子ノードを複数の並列実行領域に分割することとを特徴とするコンパイラ。  
【請求項11】 前記第1のコードスケジューラは、基本ブロックを単位として見積もられた実行時間に基づいて、さらに、前記基本ブロック内のプログラム部分におけるクリティカルパス長と、当該プログラム部分の平均並列度とを取得する、請求項10に記載のコンパイラ。  
【請求項12】 前記実行時間見積部は、  
前記子ノードにおいて実行可能な並列度ごとに、当該子ノードを並列実行した場合の実行時間の最大値を求め、各並列度における前記実行時間の最大値のうちの特定の値を前記条件分岐部分を並列実行する場合における実行時間と見積もる、請求項10に記載のコンパイラ。  
【請求項13】 前記実行時間見積部は、実行時間の判断に先立って、前記子ノードを構成する前記基本ブロックの命令レベルでの依存関係に基づいて、前記基本ブロックの実行時間に関する情報を修正する、請求項10に記載のコンパイラ。  
【請求項14】 コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、  
処理対象であるプログラムに対し、基本ブロックを単位として実行時間を見積もる処理と、  
前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成する処理と、  
前記ネスト木のノードが条件分岐を伴う場合に、前記基

本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合のどちらが前記プログラムの当該ノード部分における実行時間が短くなるかを判断する処理と、  
並列実行する方が実行時間が短いと判断した場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いと判断した場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項15】 コンピュータに、

処理対象であるプログラムに対し、基本ブロックを単位として実行時間を見積もる処理と、前記基本ブロックの接続関係を入れ子構造で表すネスト木を生成する処理と、前記ネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、当該プログラムの条件分岐部分を並列実行する場合のどちらが前記プログラムの当該ノード部分における実行時間が短くなるかを判断する処理と、並列実行する方が実行時間が短いと判断した場合は、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いと判断した場合は、当該ノードの複数の子ノードを複数の並列実行領域に分割する処理とを実行させるプログラムを記憶する記憶手段と、  
前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】 本発明は、コンピュータプログラムの最適化方法に関し、特にプログラムの条件分岐部分を並列実行に書き換える最適化を効果的に行うためにプログラムの最適化領域を編集する方法に関する。

【0002】

【従来の技術】 通常、プログラミング言語で記述されたプログラムのソースコードをコンパイルする際、コンピュータにおける実行速度の向上を図るために、当該プログラムの最適化を行っている。最適化の手法には種々の方法があるが、米国インテル社及び米国ヒューレット・パッカード社によるIA-64のアーキテクチャに対応したCPUのように、VLW (Very Long Instruction Word) による並列処理、すなわち、プレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能なプロセッサ上では、分岐命令を削除して、分岐先命令群を相補的なプレディケートを付けて並列実行することができる（このように命令群を変換することを、以下では、IF変換と呼ぶ）。このIF変換を行うことにより、命令数を減少したり、分岐予測の失敗を回避したり

(4)

特開 2002-116916

することができるため、プログラムの実行効率を向上させることが可能である。

【0003】しかし、IF変換は、この変換を行う命令の選び方によっては、かえって実行効率を低下させてしまう場合がある。その原因は、並列度のハードウェア限界の超過、レジスタプレッシャーの上昇、分岐先命令におけるクリティカルパス長のバランスの悪さ、実行可能性が低い命令の挿入などである。これら全ての原因を考慮に入れてプログラムの実行効率の高低を厳密に判断するには、各分岐命令に関して、IF変換する場合としない場合のそれぞれについてコードスケジュールを実行し、実際の命令サイクル数を見積もり、比較する必要がある。

【0004】しかしながら、プログラム中の全ての分岐命令に対して、この命令サイクル数の見積もり及び比較を行うとすると、組み合わせの数が膨大となるため、現実的な計算時間で終了することができない。そのため、最適化を実行する領域（以下、この領域をハイパーブロックと称し、図示する場合には四辺形などを用いて示す）を適切に選択する必要がある。

【0005】そこで従来は、IF変換を行うかどうかの判断を現実的な時間内で処理するため、（１）実行可能性が最も高いと予想される実行パス（以下、メイントレースと呼ぶ）中の分岐命令のみを必要に応じてIF変換する方法や、（２）一旦、全ての分岐命令をIF変換し最適化を施した後、リストスケジューリングの過程で必要に応じてIF変換の逆方向の変換（以下、逆IF変換と呼ぶ）を実行して分岐命令を再生する方法を探っていた。

【0006】これらの従来技術について、さらに説明する。

（１）メイントレース中の分岐命令のみを必要に応じてIF変換する従来技術としては、文献1  
S. A. Mahlke, R. E. Hank, R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock" in Proceedings of the 25th International Symposium on Microarchitecture, pp. 45-54, December 1992. に開示された技術がある。文献1に記載された従来技術は、どの領域を並列実行し、IF変換を実行することで性能向上が得られるかという課題について、発見的手法による一つの解を与えている。文献1によれば、まずメイントレースを特定し、このパスに対して無条件でIF変換を実行する。次に、メイントレース以外のパス（以下、サブトレースと呼ぶ）それぞれについて、同じ並列実行領域に含めるかどうかを判断し、段階的にIF変換を実行する領域を増大させてゆく。ある分岐命令に対してIF変換を実行するかどうかは、以下の四つの要因を考慮して判断する。

1. サブトレースにパイプラインを乱す命令があるかどうか。

2. メイントレースに対するサブトレースの実行確率。  
3. メイントレースに対するサブトレースの機械語命令数の比。

4. ハードウェアの並列実行能力の限界。

この方法によれば、メイントレースの分岐命令の数を $n$ とすると、 $n$ に比例する程度の計算量で、IF変換を行うかどうかの判断を終了することができる。

【0007】（２）一旦、全ての分岐命令をIF変換し最適化を施した後、リストスケジューリングの過程で必要に応じて逆IF変換を実行して分岐命令を再生する従来技術としては、文献2

D. I. August, W. W. Hwu, S. A. Mahlke, "A Framework for Balancing Control Flow and Predication" in Proceedings of the 30th International Symposium on Microarchitecture, December 1997. に開示された技術がある。文献2に記載された従来技術によれば、まず、プログラム全体を一つの並列実行領域とし、全ての分岐命令に対してIF変換を行う。そして、IF変換後のプログラムに対して種々の最適化を施し、この後、選択的に逆IF変換を実行することによって、結果として選択的に分岐命令をIF変換した状態を作る方法である。この方法は、コードスケジューラと協業して、各分岐命令に関して、逆IF変換した場合としない場合のそれぞれについて実行サイクル数を求める。そして、どちらの実行性能が高いかによって逆IF変換するかどうかを決定する。ただし、この方法を関数内の全ての分岐命令に対して適用すると、IF変換を行うかどうかを判断する場合と同様に、逆IF変換の対象となる命令群の組み合わせの数が膨大となる。そこで、リストスケジューラと協業し、クリティカルパスをスケジュールする場合にのみ逆IF変換を試みることによって、計算量を抑えている。文献2においては、分岐命令の数を $n$ とすると、 $2n$ 回程度スケジューリングを行う手法を提案している。なお、クリティカルパスとは、プログラム中の特定の範囲において、並列化できない一連の命令列のうちで最も長い命令列である。

【0008】

【発明が解決しようとする課題】しかし、上述した従来技術において、（１）メイントレース中の分岐命令のみを必要に応じてIF変換する技術は、メイントレースの実行効率を高めることはできるが、サブトレースの実行効率については特に考慮されていない。したがって、必ずしもプログラム全体の実行効率を向上させられるとは限らない。また、特に実行可能性の高いパスが無く、メイントレースを厳密に特定できない場合には、どのパスに対してIF変換を行うかを決定することが困難である。そして、何らかの基準に基づいていずれかのパスに対してIF変換を行ったとしても、他のパスの実行可能性も十分に高いため、プログラムの実行効率を十分に向上させることができない。

【0009】また、(2)一旦、全ての分岐命令をIF変換し最適化を施した後、リストスケジューリングの過程で必要に応じて逆IF変換を実行して分岐命令を再生する技術は、リストスケジューリングの過程で逆IF変換の対象となるパスを選択しているが、依然として、所定のパスに対してIF変換を行うかどうか(この場合は逆IF変換により分岐命令を再生するかどうか)を判断している。そして、この判断は、IF変換を行った場合と行わない場合とのそれぞれに対してコードスケジューリングを行って、比較するという枠組みであるため、計算量が大きい。すなわち、この従来技術においても、IF変換を行うかどうかを判断する場合と同様に、原理的に、IF変換を適用する領域の大きさと、コンパイルの速度との間にトレードオフが存在する。

【0010】以上のように、これらの従来技術は、現実的な汎用コンパイラ、特にJavaにおけるJust In Time Compilerなどのようにコンパイル時間に制約がある言語処理系では、プログラム全体の十分な実行効率の向上を図ることができなかった。また、実際の汎用の言語処理系では、十分な実行時情報が得られないため、メイントレースのような最適化すべきパスを厳密に特定することは困難であり、この問題は一層深刻であった。

【0011】そこで、本発明は、限られた時間で、広い領域をハイパーブロックに分割することにより、ある程度実行可能性が高い多くのパスにおける実行効率を向上させることが可能な最適化方法を提供することを目的とする。

【0012】

【課題を解決するための手段】かかる目的のもと、本発明は、動的計画法に基づいた方法で、プログラムの所定の領域全体(命令列a)を並列実行した場合の最短処理時間の見積もりを、その領域の一部分(命令列b、cなど)の最短処理時間の見積もりから再帰的に計算する。そして、命令列aの実行効率と命令列b、cを逐次に実行した場合の実行効率とを比較し、命令列aの実行効率の方が悪くなる場合は、命令列b及び命令列cを独立したハイパーブロックにする。これにより、プログラムにおける最適化したい領域全体を、複数のハイパーブロックに適切に分割する。

【0013】これを実現する本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もるステップと、この基本ブロックの接続関係を入れ子構造で表すネスト木を生成するステップと、このネスト木のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、並列実行する場合の、このプログラムのこのノード部分における実行時間を見積もるステップと、この見積

もりにより、並列実行する方が実行時間が短い場合は、このノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短い場合は、このノードの複数の子ノードを複数の並列実行領域に分割するステップとを含むことを特徴とする。

【0014】この最適化方法は、特にプレディケート付き命令が実行可能でかつ命令レベルの並列実行が可能な計算機にてプログラムを実行する場合に、条件分岐をそのまま実行するよりも命令レベルで並列実行する方が高速に処理できるならば、この条件分岐部分を並列実行するように書き換えることができる。

【0015】ここで、この基本ブロックを単位としてプログラムの実行時間を見積もるステップは、前記基本ブロックを単位として見積もられた実行時間に基づいて、さらに、この基本ブロック内のプログラム部分におけるクリティカルパス長と、このプログラム部分の平均並列度とを取得するステップを含む。また、この基本ブロックを、取得したクリティカルパス長及び平均並列度を辺とする矩形で表現し、さらに、この矩形を、クリティカルパス長に対応する辺がこのクリティカルパス長を下回らない範囲で変形可能とすることができる。基本ブロックについて取得されるクリティカルパス長は、この基本ブロック内部の命令列のうちで依存関係により逐次実行する必要のある最も長い部分の長さである。また、平均並列度は、基本ブロックに含まれる全ての命令の数(延べ実行時間に相当)をクリティカルパス長で割った値である。すなわち、基本ブロックの命令列を、クリティカルパス長を保持したまま並列に実行する場合に、どれだけの並列度が必要かを示す。このように平均並列度を定めた上で、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似する。

【0016】また、このネスト木を生成するステップは、基本ブロック間の依存関係を表す依存グラフを生成するステップと、この依存グラフから冗長な枝を取り除いた先行制約グラフを生成するステップと、この先行制約グラフのノードの接続関係を入れ子構造で表現することによりこのネスト木を生成するステップとを含む。ネスト木のノードは、プログラムの基本ブロックと、シリーズスートと、パラレルスートとからなる。シリーズスートは基本ブロックまたはスートが依存関係を有して直列に接続されたものであり、パラレルスートは基本ブロックまたは他のスートが依存関係を持たずに並列に並んだものである。すなわち、基本ブロックまたはスートをシリーズスートまたはパラレルスートの形で括っていくことにより、このプログラムの入れ子構造を表現する。

【0017】また、ここで、条件分岐部分の実行時間を判断するステップは、この子ノードにおいて実行可能な並列度ごとに、この子ノードを並列実行した場合の実行時間の最大値を求めるステップと、各並列度における実

行時間の最大値のうち特定の値をこの条件分岐部分を並列実行する場合における実行時間と見積もるステップとを含む。

【0018】この条件分岐部分の実行時間を判断するステップは、実行時間の判断に先立って、この子ノードを構成する基本ブロックの命令レベルでの依存関係に基づいて、この基本ブロックの実行時間に関する情報を修正するステップを含む。具体的には、シリーズストを構成する基本ブロックにおいて、前に位置する基本ブロック内部における最後の命令以外の命令と、後ろに位置する基本ブロック内部における最初の命令とが依存関係を有する場合、このシリーズストは基本ブロックを単に直列に連結するよりも短い長さ（このシリーズストのクリティカルパス長）で実行できることとなる。

【0019】さらにここで、このプログラムの並列実行領域を決定するステップは、この複数の子ノードを複数の並列実行領域に分割する場合に、ハードウェアが持つ並列度で各子ノードを並列実行した場合における各子ノードの実行時間を比較するステップと、この実行時間が最も短い子ノードを残して、他の子ノードを独立した並列実行領域とするステップとを含む。

【0020】また、本発明は、これらの最適化方法を、コンピュータによるプログラムのコンパイルにおいて実行させるコンピュータプログラムとして作成し、このコンピュータプログラムを格納した記憶媒体や、このコンピュータプログラムを伝送する伝送装置として提供することができる。

【0021】さらに、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、処理対象である前記プログラムに対し、基本ブロックを単位として実行時間を見積もる第1のコードスケジューラと、この基本ブロックをまとめて並列実行領域であるハイパーブロックを生成するハイパーブロック生成部と、このプログラムにおける所定の領域を実行した場合の処理時間を見積もることにより、このハイパーブロック生成部によるこのハイパーブロックの生成を支援する実行時間見積部と、生成されたハイパーブロックごとにコードスケジューリングを行う第2のコードスケジューラとを備え、この実行時間見積部は、この基本ブロックの接続関係を入れ子構造で表すネスト木の所定のノードが条件分岐を伴う場合に、前記基本ブロックを単位として見積もられた実行時間に基づいて、条件分岐のまま実行する場合と、このプログラムの条件分岐部分を並列実行する場合の、当該プログラムの当該ノード部分における実行時間を見積もり、このハイパーブロック生成部は、実行時間見積部の見積もりにより、並列実行する方が実行時間が短いノードに関して、当該ノード部分を一まとまりの並列実行領域とし、条件分岐のまま実行する方が実行時間が短いノードに関して、当該ノードの複数の子ノ

ードを複数の並列実行領域に分割することを特徴とする。

【0022】ここで、この第1のコードスケジューラは、基本ブロックを単位として見積もられた実行時間に基づいて、さらに、この基本ブロック内のプログラム部分におけるクリティカルパス長と、このプログラム部分の平均並列度とを取得する。また、この実行時間見積部は、実行時間の判断に先立って、この子ノードを構成する基本ブロックの命令レベルでの依存関係に基づいて、この基本ブロックの実行時間に関する情報を修正する。

【0023】

【発明の実施の形態】以下、添付図面に示す実施の形態に基づいて、この発明を詳細に説明する。まず、本発明の概要を説明する。上述したように、本発明は、プログラムのコンパイル時における最適化において、プログラムの所定の領域を適切なハイパーブロックに分割する。

【0024】具体的には、まず、処理対象であるプログラムを変形可能に定義された基本ブロックに分割し、基本ブロック間の制御依存関係及びデータ依存関係に基づいて、PDG (Program Dependence Graph: 依存グラフ) を作る。ここで、基本ブロックとは、ストレートコード、すなわちコントロールフローが途中に入ることもなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。本発明では、後述のように基本ブロックを構成する命令列のクリティカルパス長を下回らない範囲で、変形可能に基本ブロックを定義する。次に、作成されたPDGをシリーズパラレルグラフに変換し、このシリーズパラレルグラフを基本ブロック間の先行制約グラフとして用いる。そして、このシリーズパラレルグラフにおける各基本ブロック内に対してコードスケジューリングを実行し、メモリ依存を超える投機的命令移動などの最適化を実行する。なお、シリーズパラレルグラフの定義については後述する。

【0025】次に、シリーズパラレルグラフにおける各基本ブロックに関して、クリティカルパス長、平均の並列度、消費するハードウェアリソースなどの情報を求めておき、これらの値を基に、基本ブロックの領域全体の最短処理時間を再帰的に見積もる。この過程で、所定の領域について、並列実行より逐次実行の方が処理速度が速いと判断された場合は、当該領域に含まれる部分領域それぞれを独立したハイパーブロックとし、当該ハイパーブロックの中の全ての条件分岐命令をIF変換する。これらの仕組みは、動的計画法を用いることにより、基本ブロックの数に比例する計算時間で終了することができる。

【0026】図1は、本発明の実施の形態におけるコンパイラの構成を説明する図である。図1に示すコンパイラは、JavaのJust In Time Compilerである。本実施の形態では、本発明をJust In Time Compilerに適用する場合を例として説明するが、他の種々のプログラム

言語で記述されたプログラムに対するコンパイラに適用できることは言うまでもない。

【0027】図1を参照すると、本実施の形態におけるコンパイラは、フロントエンド最適化部10と、命令レベル並列化最適化部20と、ネイティブコード生成部30とを備える。フロントエンド最適化部10は、Javaにおける処理対象のプログラムのバイトコードを入力し、バイトコードレベルでの最適化を行うバイトコード最適化部11と、バイトコード最適化部11により最適化を施されたバイトコード（四つ組中間コード）に対して四つ組中間コードレベルでの最適化を行う四つ組中間コード最適化部12とを備える。また、四つ組中間コード最適化部12では、処理対象であるプログラムを基本ブロックに分割する。

【0028】命令レベル並列化最適化部20は、フロントエンド最適化部10により四つ組中間コードレベルまでの最適化が行われたプログラムに対して、基本ブロックレベルでのコードスケジューリングを行う基本ブロック・コードスケジューラ21と、当該基本ブロック間の依存関係に基づいてPDGを作成し、このPDGをハイパーブロックに適切に分割するためのハイパーブロック生成部22及び実行時間見積部23と、ハイパーブロックに分割されたプログラム全体に対するコードスケジューリングを行うグローバル・コードスケジューラ24とを備える。

【0029】ネイティブコード生成部30は、命令レベル並列化最適化部20によりコードスケジューリングが行われたプログラムをネイティブコードに変換するコード生成部31と、コード生成部31により生成されたネイティブコードに対して命令レベルでのコードスケジューリングを行う命令レベル・コードスケジューラ32とを備える。

【0030】以上の構成のうち、フロントエンド最適化部10とネイティブコード生成部30とは、従来のJust In Time Compilerにおけるフロントエンド最適化部10及びネイティブコード生成部30と同様である。したがって、本実施の形態は、命令レベル並列化最適化部20において、基本ブロックレベルでのコードスケジューリングを行い、ハイパーブロック生成部22及び実行時間見積部23において、命令列の実行時間を見積もりながらハイパーブロックを生成した上で、全体的なコードスケジューリングを行う点に特徴がある。また、図1に示すJust In Time Compiler以外の、他のプログラミング言語におけるコンパイラにおいても、ソースプログラムを入力して構文解析を行い中間コードを生成した後に、図1の命令レベル並列化最適化部20に相当する機能ブロックにより、同様の命令レベルでの並列化による最適化処理を行うことができる。

【0031】なお、図1に示す各構成要素は、コンピュータプログラムにより制御されたCPUにて実現される

仮想的なソフトウェアブロックである。CPUを制御する当該コンピュータプログラムはCD-ROMやフロッピー（登録商標）ディスクなどの記憶媒体に格納したり、ネットワークを介して伝送したりすることにより提供される。

【0032】次に、命令レベル並列化最適化部20の動作について説明する。本実施の形態は、処理対象であるプログラム中の命令間の制御依存関係及びデータ依存関係を示すPDG（Program Dependence Graph）をシリーズパラレルグラフに変換し、命令列の実行時間の見積もりを再帰的に行うことにより、適切なハイパーブロックを生成する。以下では、命令レベル並列化最適化部20の動作について、まず概略的に説明し、次に具体的な手法について説明する。

【0033】まず、本実施の形態で用いる基本ブロックについて詳細に説明する。図39は、処理対象のプログラムの構成を基本ブロックで表現した例を示す図である。図39に示した基本ブロックD、Eの組のように互いに依存がない基本ブロックの集合（パラレルスートと呼ぶ、定義は後述）の場合、並列に実行しても、逐次に行っても、プログラムの正しさは保証されているので、どちらで実行されると領域全体の結果が良いかどうかを決定する必要がある。この決定は、単純には、領域中の全てのパラレルスートを、逐次、並列のどちらかで実行したときの全ての組み合わせに対して、全体の処理時間と必要な並列度を求めて比較する方法で実現できる。その際、全体の処理時間と必要な並列度を求めるには、単純には、パラレルスートの実行時間を構成要素の実行時間の合計 $d$ と論理的な並列度の最大値 $w$ とで表すことができる。図40は、パラレルスートの実行時間を構成要素のパラメータ（ $d$ 、 $w$ ）で表現できることを説明する図である。

【0034】しかし、この方法には、次の三つの問題点がある。1. 基本ブロックは、上述したように、ストレートコード、すなわちコントロールフローが途中に入ることなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。したがって、本来的には変形不能の矩形領域ではなく、実際には（ある制約の下で）柔軟に移動可能な命令の集合であるが、上の基本ブロックの定義ではこの点が考慮されていない。

2. スートを構成する際に図40に示した「すきま」ができるため、処理時間と並列度の見積もり精度が悪い。3. 領域中のパラレルスートを並列処理する場合と逐次処理する場合の全ての組み合わせを尽くす方法で実行時間を見積もるため、計算時間が基本ブロックの数に応じて指数関数的に増加する。

【0035】そこで、本実施の形態では、各基本ブロックを、クリティカルパス長を下回らないという前提の下で、縦方向（処理時間が伸びる方向）に伸ばせるようにモデル化する。このようにモデル化された基本ブロック

によるパラレルスートの例を図41に示す。図41に示すように、パラレルスートは、基本ブロックDと基本ブロックEとを変形させながら処理時間と平均並列度を見積もることができる。これにより、より実際のプログラムの構成に即した見積もりを行うことが可能となる。また、基本ブロックを変形可能としたことにより、上述したスートにおける「すきま」を解消することもできる。さらに、基本ブロックの横幅である平均並列度は、基本ブロックにおける命令列の延べ実行時間とクリティカルパス長に基づいて単純なモデル化により算出された値であるため、図42に示すような基本ブロック内部における隙間も、複数の基本ブロックを扱うことによってある程度相殺されることが期待される。

【0036】次に、このモデルに従った最短処理時間の見積もりを、基本ブロックの数に応じて計算時間が指数関数的に増加することを回避して高速に実現するために、動的計画法を使った実現方法を説明する。本実施の形態では、所定の領域の最短処理時間の見積もりを、当該領域における部分領域の最短処理時間の見積もりから再帰的に求めていく。例えば、図39の基本ブロックD、Eの処理時間の見積もりから、それらで構成されるパラレルスートの実行時間を見積もる。この再帰処理の1ステップを高速に実現することで、全体として高速な実行を可能とする。この処理を実現するには、基本ブロックだけでなく、パラレルスート、実行依存がある基本ブロックの集合（シリーズスートと呼ぶ、定義は後述）を、変形可能な矩形領域とみなせなければならない。これを満たすため、基本ブロック、パラレルスート、シリーズスートに関して、ハードウェアの並列度 $W$ に対する、それぞれの可能な並列度 $w$  ( $1 \leq w \leq W$ ) に対する最短処理時間の見積もりデータを再帰的に計算していく。

【0037】基本ブロック、パラレルスート、シリーズスートの各場合について、上記のデータを見積もる方法を簡単に述べる。基本ブロックに対しては、基本ブロック・コードスケジューラ21によるコードスケジューリングを実行し、延べ実行時間を求める。そして、この延べ実行時間を並列度 $w$ で割った値を、当該基本ブロックの最短処理時間の見積もりとする。ただし、本実施の形態における基本ブロックの定義より、クリティカルパス長は下回らないものとする。シリーズスートに対しては、構成要素の並列度 $w$ における処理時間を、単純に加算する。パラレルスートに対しては、図40のイメージどおり、 $1 \leq w \leq W$ における各並列度 $w$ について、 $w_1 + w_2 = w$ の条件を満たしながら $w_1$ を変化させ、各構成要素を並列実行した場合の実行時間の最大値を求め、得られた最大値の最小値を当該スートの最短処理時間の見積もりとする。これらの手続き、すなわち再帰の1ステップは、最長でも $W^2/2$ に比例する計算時間で終了する。結局、基本ブロックの数を $n$ とすると、 $n \times W^2$ に比例した計算時間で全体の処理が終了する。

【0038】次に、命令レベル並列化最適化部20による動作の具体的な手法について説明する。本実施の形態では、上記の動的計画法を使ったハイパーブロックの生成手法として、基本ブロックの構成のみを用いた第1の手法と、基本ブロック内の命令レベルの依存関係をも考慮した第2の手法とを提案する。

【0039】第1の手法は、基本ブロックを、論理的な並列度の平均とクリティカルパス長で表される矩形領域にモデル化する。複数の基本ブロックにおける逐次実行時間の見積もりは、クリティカルパスの処理時間の和を下回らないとする。また、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似する。第2の手法は、基本ブロックを、論理的な並列度の平均とクリティカルパス長で表される矩形領域にモデル化すると共に、当該基本ブロック内の命令間における依存関係 (Dependence Path) の情報を持たせる。複数の基本ブロックの逐次実行時間の見積もりは、Dependence Pathを再構成してクリティカルパス長を再計算し、その値を下回らないとする。複数の基本ブロックを並列に実行した時必要とされる並列度は、第1の手法と同様に扱う。基本ブロックを、平均並列度とクリティカルパス長で表される矩形領域に近似することにより、複数の基本ブロックを並列に実行した場合に必要とされる並列度は、基本ブロックの平均並列度の線形和になると近似することができ、これにより、高速動作を実現することができる。以下、手法ごとに詳細に説明する。なお、以下の説明において用いられる記号を図7において定義する。図7の各定義において、nodeはノード一般を示す。したがって、以下の説明において、特に所定のノードにおける子ノードを指す場合には、nodeの代わりにchildなどと記述する場合もある。

【0040】【第1の手法】命令レベル並列化最適化部20は、初期的に、フロントエンド最適化部10により前処理としての最適化を施されたプログラム（中間コード）を入力し、実行頻度情報やプログラムの構造に基づいて並列実行する領域を決定する。例えば、ある程度以上の実行頻度を持ち、ループを含まない領域を並列実行する領域とすることができる。ここでは、図2に示したプログラム領域を処理対象とする。図2は、処理対象であるプログラムの最適化処理を行う領域の制御フローグラフと当該部分の命令列のリストである。

【0041】基本ブロック・コードスケジューラ21は、当該領域内の基本ブロック毎に、あいまいなメモリ依存の解消を伴うコードスケジューリングを行う。これにより、各基本ブロック（図7におけるnodeとする）について、node.clと、node.allとを見積もることができる。

【0042】図3は、図2に示したプログラム領域において、node.clを縦の長さとし、node.allを面積とする



矩形領域で基本ブロックを表した図である。また、基本ブロックの横の長さ、すなわち面積 (node.all) を縦の長さ (node.cl) で割った値を平均並列度と呼ぶ。例えば、図2によると、ノードAには、「imove a, 1」「imove b, 1」「imove c, 3」「iadd a, 1」「isub b, 1」「cmpjmp-eq a, 1, C」という6個の命令が含まれる。このうち、「imove a, 1」「iadd a, 1」「cmpjmp-eq a, 1, C」は依存関係があるので、並列に実行できない。全部で6個の命令のうち、3個が逐次実行されるので、これがクリティカルパス長となる。したがって、図3に示すようにノードAは、縦の長さが3、横の長さが2 ( $= 6 / 3$ ) の矩形として表現される。

【0043】次に、ハイパーブロック生成部22は、図3に示す基本ブロック間の制御依存関係及びデータ依存関係を表すPDGを作成する。そして、作成されたPDGから先行制約グラフとして冗長な枝を取り除き、シリーズパラレルグラフに変換する。ここで、シリーズパラレルグラフとは、次の三つの条件を満たすグラフである。

- (1) 単独のノードはシリーズパラレルグラフである。
- (2) シリーズパラレルグラフを二つ直列に接続したものはシリーズパラレルグラフである。
- (3) シリーズパラレルグラフを二つ並列に接続したものはシリーズパラレルグラフである。

また、シリーズパラレルグラフの構成は、シリーズスートとパラレルスートの2種類の部分に分けることができる。シリーズスートとは、直列に連結（この連結をシリーズコネクションと呼ぶ）された、依存関係のある一連のシリーズスート、パラレルスート及びノードの集合である。また、パラレルスートとは、並列に連結（この連結をパラレルコネクションと呼ぶ）された、互いに依存関係のないシリーズスート、パラレルスート及びノードの集合である。図8は、PDGをシリーズパラレルグラフに変換するアルゴリズムを示す疑似プログラムである。図示のアルゴリズムによる手続きは、現実的にはPDGのノードの数に比例した計算時間で終了する。

【0044】図4は、図3から作成されたPDGを示す図、図5は、図4のPDGから変換されたシリーズパラレルグラフを示す図である。図4、5を参照すると、例えば、図4におけるノードAからノードDへのデータ依存は、ノードAからノードCへの依存とノードCからノードDへの依存が存在するため、冗長とみなされ、図5に示すように取り除かれている。

【0045】次に、ハイパーブロック生成部22は、上述のシリーズパラレルグラフから、シリーズスートとパラレルスートの入れ子関係を表すシリーズパラレルネスト木を生成する。シリーズパラレルネスト木とは、次のように定義されるノードとエッジとを持つ木構造である。

ノード：シリーズパラレルグラフ中の全てのシリーズス

ートまたは全てのパラレルスートまたは全てのノードの集合。

エッジ：所定のシリーズスートに対し、シリーズコネクションのみで連結された一連のシリーズスート、パラレルスートまたはノードがある場合、この所定のシリーズスートから当該一連のスートまたはノードに対して張られるエッジ。または、所定のパラレルスートに対し、パラレルコネクションのみで連結された一連のシリーズスート、パラレルスートまたはノードがある場合、この所定のパラレルスートからその一連のスートまたはノードに対して張られるエッジ。図9は、シリーズパラレルグラフからシリーズパラレルネスト木を生成するアルゴリズムを示す疑似プログラムである。図示のアルゴリズムによる手続きは、実践的にはシリーズパラレルグラフのノード数に比例した時間で終了する。ただし、図示のアルゴリズムは、単純のため、パラレルスートが3以上の葉ノードを持つことを許しているが、葉ノードを2つだけ持つように変更することも容易に可能である。図6は、図5のシリーズパラレルグラフから生成されたシリーズパラレルネスト木を示す図である。なお、図5、6において、実線で示したノードはシリーズスートを示し、破線で示したノードはパラレルスートを示す。

【0046】次に、ハイパーブロック生成部22は、実行時間見積部23を用いて、シリーズパラレルネスト木における各ノードの実行時間を再帰的に見積もり、その結果に基づいてハイパーブロック選択処理を実行する（図10参照）。このハイパーブロック選択処理により、シリーズパラレルネスト木の各ノードに、独立したハイパーブロックとして扱うか否かを示す情報が付される。以下、ハイパーブロック選択処理について、詳細に説明する。

【0047】図11は、ハイパーブロック選択処理の全体的な動作の流れを示すフローチャートである。また、図12は、図11に対応する動作のアルゴリズムを示す疑似プログラムである。図11を参照すると、まず、シリーズパラレルネスト木のノードの一つを処理対象とし（ステップ1101）、当該ノードの属性を調べ、パラレルスートか、シリーズスートか、基本ブロック（単一ノード）かを判断する（ステップ1102）。そして、当該ノードがパラレルスートであれば、実行時間見積部23に処理を渡して当該パラレルスートの実行時間見積処理を行う（ステップ1103）。また、当該ノードがシリーズスートであれば、当該ハイパーブロック選択処理を当該ノードの子ノード（ここでは子ノード1、子ノード2の二つ）に再帰的に適用した後（ステップ1104）、当該シリーズスートの実行時間評価処理を行う（ステップ1105）。さらに、当該ノードが基本ブロック（単一ノード）であれば、単一ノードの実行時間評価処理を行う（ステップ1106）。

【0048】ステップ1103によるパラレルスートの

実行時間見積処理では、並列度 $W$ のときの所定のパラレルスト( $x$ )の処理時間を求めたい場合、パラレルスト( $x$ )を構成するスト( $y$ ,  $z$ )について、 $W = w_1 + w_2$ の関係で $w_1$ を変化させながら、並列度 $w_1$ のときのスト( $y$ )の最短処理時間及び並列度 $w_2$ のときのスト( $z$ )の最短処理時間の最大値を求める。そして、 $w_1$ の値ごとに得られた最大値のうちの最も小さい値を、並列度 $W$ のときのパラレルスト( $x$ )の最短処理時間として見積もる。ここで、パラレルストは、各ノードをまとめて一つのハイパーブロックを構成することによって、処理時間を長くしてしまうことがあり得る。そこで、パラレルスト中のノードをまとめてハイパーブロックを構成するか否かの判断を行う。具体的には、分岐命令削減による利得を並列実行のオーバーヘッドが下回っているかどうかを判断する。そして、構成しないと判断された場合は、各ノードを分けて別のハイパーブロックを構成するという意味の印を付ける(`node.hyper_block = TRUE` (図7))。この際、どちらのストを独立したハイパーブロックにするかの判断は、状況に応じて種々の手法で行うことができる。例えば、単純にクリティカルパスが長い方を独立したハイパーブロックとしても良いし、実行確率がわかっている場合は、実行確率の低い方を独立したハイパーブロックとすることもできる。

【0049】図13は、図11におけるステップ1103のパラレルストに対する実行時間見積処理の内容を説明するフローチャートである。図13を参照すると、まず、処理対象であるノードの子ノード(ここでは子ノード1、子ノード2の二つ)に当該ハイパーブロック選択処理を再帰的に適用する(ステップ1301)。そして、パラレルストの実行時間評価処理(ステップ1302)及び利得見積処理(ステップ1303)を行った後、得られた利得の値が0よりも大きいのか否かを判断する(ステップ1304)。利得が0以下である場合は、ノード選択処理を行う(ステップ1305)。利得が0よりも大きい場合は、子ノード1と子ノード2の面積(すなわち、`node.all`: 当該ノードの延べ実行時間)の和を親ノード(すなわち、初めに処理対象としたノード)の面積とする(ステップ1306)。

【0050】図14は、図13におけるステップ1302の実行時間評価処理の内容を説明するフローチャートである。また、図15は、図14の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図14を参照すると、まず、この処理の中でのみ有効な自由変数 $w_1$ の値を1に初期化する(ステップ1401)。また、この処理の中でのみ有効な自由変数 $min$ の値を無限大に初期化する(ステップ1402)。そして、変数 $w_1$ とハードウェアの並列度 $W$ とを比較し、同じかどうか調べる(ステップ1403)。変数 $w_1$ が並列度 $W$ と同じでなければ、次に、この処理の中でのみ有効な自由

変数 $w_2$ を0に初期化し(ステップ1404)、この変数 $w_2$ と変数 $w_1$ とを比較して同じかどうか調べる(ステップ1405)。変数 $w_1$ と変数 $w_2$ とが同じでない場合、次に、子ノード1が並列度 $w_2$ のときの最短処理時間と、子ノード2が並列度( $w_1 - w_2$ )のときの最短処理時間とを比較し、大きい方の値を求める(ステップ1406)。そして、得られた値と、変数 $min$ の値とを比較し、小さい方の値を変数 $min$ の新たな値とする(ステップ1407)。この後、変数 $w_2$ の値を1増加し(ステップ1408)、ステップ1405の比較に戻る。一方、変数 $w_1$ と変数 $w_2$ とが同じである場合は、現時点での変数 $min$ の値を親ノード(すなわち、初めに処理対象としたノード)が並列度 $w_1$ のときの最短処理時間とする(ステップ1409)。そして、変数 $w_1$ の値を1増加し(ステップ1410)、ステップ1402に戻る。そして、ステップ1403の比較において、変数 $w_1$ と並列度 $W$ とが同じであれば、処理を終了する(ステップ1403)。

【0051】以上の動作により、並列度 $W$ のときのパラレルストの最短処理時間の見積もりが得られる。実際のCPU(ハードウェア)における並列度 $W$ は一桁程度の小さな数値なので、この実行時間評価処理はノード数に比例する時間で終了することができる。

【0052】図16は、図13におけるステップ1303の利得見積処理の内容を説明するフローチャートである。また、図17は、図16の利得見積処理の動作アルゴリズムを示す疑似プログラムである。図16を参照すると、図14、15に示した実行時間評価処理により得られたパラレルストの最短処理時間の見積もりに基づいて、子ノード1と子ノード2とを条件分岐としてそのまま実行した場合の処理時間と、この条件分岐をブレイク付き命令で並列実行した場合の処理時間との差を利得として求める(ステップ1601)。

【0053】図18は、図13におけるステップ1305のノード選択処理の内容を説明するフローチャートである。また、図19は、図18のノード選択処理の動作アルゴリズムを示す疑似プログラムである。図18を参照すると、まず、ハードウェアの並列度 $W$ のときの子ノード1及び子ノード2の最短処理時間を比較する(ステップ1801)。子ノード1の最短処理時間が子ノード2の最短処理時間以下である場合、子ノード2に対して当該子ノード2を独立したハイパーブロックとして実行するという意味の属性を付する(ステップ1802)。そして、子ノード1の最短処理時間の情報、依存関係を示す情報、及び延べ実行時間の情報を親ノード(すなわち、最初に処理対象としたノード)に複写する(ステップ1803)。これにより、当該親ノードは、子ノード1のみから構成されることとなる。一方、ステップ1801の判断において、子ノード1の最短処理時間の方が子ノード2の最短処理時間よりも大きい場合、子ノード

1に対して当該子ノード1を独立したハイパーブロックとして実行するという意味の属性を付する(ステップ1804)。そして、子ノード2の最短処理時間の情報、依存関係を示す情報、及び延べ実行時間の情報を親ノード(すなわち、最初に処理対象としたノード)に複写する(ステップ1805)。これにより、当該親ノードは、子ノード2のみから構成されることとなる。

【0054】以上のようにして、シリーズパラレルネスト木のパラレルスートに対する実行時間見積処理を終了する。これにより、シリーズパラレルネスト木中のパラレルスートに関して、当該パラレルスートを構成するノードを一つのハイパーブロックにまとめるか、または別のハイパーブロックとして分岐させるかを示す情報と、当該一つまたは二つのハイパーブロックにおける実行時間の見積もりが得られ、当該ハイパーブロックに付されることとなる。

【0055】図11のステップ1105によるシリーズスートの実行時間評価処理では、当該シリーズスートを構成するノードの最短処理時間を単純に加えることにより、当該シリーズスートにおける実行時間を粗く見積もる。そしてさらに、平均並列度 $w$ に対して当該実行時間が最短(最短処理時間)となるように補正する。

【0056】図20は、図11におけるステップ1105のシリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。また、図21は、図20の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図20を参照すると、まず、処理対象であるノードの延べ実行時間を子ノード(ここでは子ノード1、子ノード2の二つ)の延べ実行時間の和とする(ステップ2001)。次に、この処理の中でのみ有効な自由変数 $w$ の値を1に初期化する(ステップ2002)。そして、変数 $w$ とハードウェアの並列度 $W$ とを比較し、同じかどうか調べる(ステップ2003)。変数 $w$ が並列度 $W$ と同じでなければ、次に、処理対象であるノードの並列度 $w$ のときの最短処理時間を、子ノード1及び子ノード2の並列度 $w$ のときのそれぞれの最短処理時間の和とする(ステップ2004)。そして、処理対象であるノードの最短処理時間に変数 $w$ を乗じた値が、子ノード1及び子ノード2の延べ実行時間とを比較し、同じかどうか調べる(ステップ2005)。すなわち、変数 $w$ が処理対象であるノードの平均並列度となっているかどうかを確認する。処理対象であるノードの最短処理時間に変数 $w$ を乗じた値が、子ノード1及び子ノード2の延べ実行時間とは異なっている場合、当該処理対象であるノードの平均並列度に対する最短処理時間をステップ2004で求めた値とする(ステップ2006)。処理対象であるノードの最短処理時間に変数 $w$ を乗じた値が、子ノード1及び子ノード2の延べ実行時間と同じである場合、またはステップ2006の終了後、変数 $w$ の値を1増加し(ステップ2007)、ステップ200

3に戻る。そして、ステップ2003の比較において、変数 $w$ が並列度 $W$ と同じであれば、処理を終了する(ステップ2003)。

【0057】図11のステップ1106による単一ノードの実行時間評価処理では、クリティカルパス長を下回らない限り、計算総量を並列度で割った値で実行できると見積もる。図22は、図11におけるステップ1106の単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。また、図23は、図22の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図22を参照すると、まず、この処理の中でのみ有効な自由変数 $w$ を1に初期化する(ステップ2201)。そして、変数 $w$ とハードウェアの並列度 $W$ とを比較し、同じかどうか調べる(ステップ2202)。変数 $w$ が並列度 $W$ と同じでなければ、次に、処理対象であるノードの最短処理時間を、基本ブロック中の延べ実行時間を変数 $w$ で割った値とする(ステップ2203)。そして、得られた値が基本ブロックのクリティカルパス長を下回ったかどうかを判断する(ステップ2204)。ステップ2203で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っている場合、当該ノードの最短処理時間を基本ブロックのクリティカルパス長とする(ステップ2205)。ステップ2203で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っていない場合、またはステップ2205の終了後、変数 $w$ の値を1増加し(ステップ2206)、ステップ2202に戻る。そして、ステップ2202の比較において、変数 $w$ が並列度 $W$ と同じであれば、処理を終了する(ステップ2202)。

【0058】以上説明したパラレルスート、シリーズスート、単一ノードに対する実行時間の見積もり及び評価を、シリーズパラレルネスト木のルートノードから葉ノードへ再帰的に実行することにより、適切なハイパーブロックを生成することができる。すなわち、一つのハイパーブロックにまとめた方が実行時間が短くなるノードに関しては、一つのハイパーブロックにまとめられ、一つのハイパーブロックにまとめない方が実行時間が短くなるノードに関しては、別のハイパーブロックに分割する。

【0059】次に、この第1の手法によるハイパーブロックの生成例を説明する。ここでは、並列処理を実行する方が実行時間が短くなる例と、条件分岐を実行した方が実行時間が短くなる例とを示すため、ハードウェアの並列度が6である場合と3である場合とを例として説明する。なお、図2乃至図6に示したプログラムを処理対象とし、各命令は時間1で実行されると仮定する。また、分岐処理によるペナルティ(分岐処理を行うことにより必然的に処理に要する実行時間)を5とする。

【0060】まず、ハードウェアの並列度が6である場

合について説明する。例として、図6のシリーズパラレルネスト木において、スート5に関し、並列度Wが3である場合の実行時間の見積もりを考える。この場合、スート5は構成要素である基本ブロックD、Eが並列に接続されたパラレルスートである。したがって、図14に示したパラレルスートに対する実行時間評価処理により、基本ブロックD、Eの並列度ごとの処理時間の最大値を求める。この場合、スート5の並列度Wが3であるから、基本ブロックD、Eの並列度は、基本ブロックDの並列度が2、基本ブロックEの並列度が1である場合と、基本ブロックDの並列度が1、基本ブロックEの並列度が2である場合とが考えられる。また、図3の基本ブロックDを表す矩形の情報から、基本ブロックDに関しては、クリティカルパス長が3であり、平均並列度が1（命令を並列に実行できず、逐次実行しなければならないことを意味する）である。したがって、基本ブロックDの処理に要する時間は、並列度に関わらず3である。同様に、図3の基本ブロックEを表す矩形の情報から、基本ブロックEに関しては、クリティカルパス長が2であり、平均並列度が4である。したがって、基本ブロックDの処理に要する時間は、並列度が1のときで8、並列度が2のときで4、並列度が3のときで3、並列度が4以上では2となる。したがって、上述した基本ブロックDの並列度が2、基本ブロックEの並列度が1である場合は、基本ブロックDの実行時間が3、基本ブロックEの実行時間が8である。一方、基本ブロックDの並列度が1、基本ブロックEの並列度が2である場合は、基本ブロックDの実行時間が3、基本ブロックEの実行時間が4である。すなわち、基本ブロックD、Eを並列に実行するとすると、基本ブロックDの並列度が2、基本ブロックEの並列度が1である場合は実行時間が8となり、基本ブロックDの並列度が1、基本ブロックEの並列度が2である場合は実行時間が4となる。そこで、スート5の最短処理時間は4と見積もることができる。図24は、以上のような最短処理時間の見積もりを、図6のシリーズパラレルネスト木における各ノード（スート）に対して、並列度1～6の各場合で求めた結果を示す図である。

【0061】ここで、図6のシリーズパラレルネスト木におけるパラレルスートであるスート5及びスート2の実行時間見積処理においては、一つのハイパーブロックにまとめるか、別の独立したハイパーブロックを生成するかの判断を行う（図13参照）。例として、スート2の場合を考えると、図24を参照して、並列度6のときのスート2の最短処理時間は6となる（基本ブロックBの並列度が2でスート3の並列度が4である場合、または基本ブロックB及びスート3の並列度が共に3である場合）。これに対し、図16、17に示す利得見積処理によれば、スート3と基本ブロックBとを、条件分岐をそのまま実行した場合の処理時間は、並列度6のときの

基本ブロックBの最短処理時間が4、スート3の最短処理時間が5であるから、 $9.5 (= 5 + (4 + 5) / 2)$  である。したがって、条件分岐を行う場合の処理時間は並列処理における最短処理時間よりも大きいため、利得は0よりも大きくなる（ $9.5 - 6 = 3.5 > 0$ ）。これにより、スート2は一つのハイパーブロックにまとめられることとなる（図13、ステップ1304、1306参照）。スート2は、他のノードである基本ブロックA、Gとシリーズスートを構成しているため、最大の並列度6の場合についてののみ考察すればよいが、スート5の場合は、スート2を構成するスート3にどれだけの並列度が割り当てられるかに応じて、それぞれ利得を見積もる必要がある。煩雑になるので記載は省略するが、同様の計算を各並列度におけるスート5に対して行くと、全ての場合で利得が0よりも大きくなる。したがって、ハードウェアの並列度が6の場合は、全ての基本ブロックを一つにまとめたハイパーブロックを生成することとなる。図24において、アスタリスク（\*）の付されたスート（スート0）は、独立したハイパーブロックを構成するという意味の印である（図7におけるnode.hyper\_block = TRUE）。

【0062】次に、ハードウェアの並列度が3である場合について説明する。ハードウェアの並列度が6である場合について説明したのと同様の手法で、図6のシリーズパラレルネスト木の各ノードに対して並列度1～3の各場合で求めた最短処理時間の見積もりを図25に示す。まず、スート2について考える。スート2を並列実行する場合、最短処理時間は12となる（基本ブロックBの並列度が1でスート3の並列度が2である場合）。これに対し、図16、17に示す利得見積処理によれば、スート3と基本ブロックBとを、条件分岐をそのまま実行した場合の処理時間は、並列度3のときの基本ブロックBの最短処理時間が4、スート3の最短処理時間が6であるから、 $10 (= 5 + (6 + 4) / 2)$  である。したがって、利得は-2（ $= 10 - 12$ ）であり、0よりも小さいので、スート2は二つのハイパーブロックに分割される（図13、ステップ1304、1305参照）。また、基本ブロックBとスート3とでは、スート3の方が最短処理時間が大きいので、スート3を独立のハイパーブロックとし、基本ブロックBは基本ブロックA及び基本ブロックGとシリーズスートを構成するハイパーブロックに含める（図18参照）。次に、スート5について考える。上述したようにスート3は、独立のハイパーブロックであり、当該スート3においてスート5は、基本ブロックCおよび基本ブロックFとシリーズスートを構成する。したがって、スート5の並列度はハードウェアの並列度3をそのまま適用できる。この場合、スート5の最短処理時間は4である（基本ブロックDの並列度が1で基本ブロックEの並列度が2である場合）。これに対し、図16、17に示す利得見積処理に

よれば、基本ブロックD、Eを、分岐条件をそのまま実行した場合の処理時間は、並列度3のときの基本ブロックD、Eとも最短処理時間が3であるから、 $8 (= 5 + (3 + 3) / 2)$ である。したがって、利得は4 ( $= 8 - 4$ )であり、0よりも大きいので、スート5は一つのハイパーブロックにまとめられる。以上の結果、基本ブロックA、B、Gが一つのハイパーブロックを形成し、基本ブロックC、D、E、Fが別の独立したハイパーブロックを形成することとなる。図25において、アスタリスク(\*)の付されたスート(スート0、3)は、独立したハイパーブロックを構成するという意味の印である(図7におけるnode.hyper\_block = TRUE)。

【0063】[第2の手法] 次に、当該基本ブロック内の命令間における依存関係(Dependence Path)の情報を持たせる第2の手法について説明する。第2の手法では、プログラムの各基本ブロックに命令レベルでの依存関係に関する情報を持たせておく。そして、基本ブロックの実行時間の見積もりの際に、この依存関係に基づいて当該基本ブロックのクリティカルパス長を再計算する。

【0064】図26を参照して具体的に説明する。図26は、図3に示した基本ブロックC、D、E、Fの内部の命令における依存関係を説明する図である。図3に示したように、基本ブロックCのクリティカルパス長は1、基本ブロックDのクリティカルパス長は3、基本ブロックEのクリティカルパス長は2、基本ブロックFのクリティカルパス長は1である。したがって、第1の手法によれば、基本ブロックC、D、E、F(図5、6のスート3に対応)の最短処理時間は5以下にはなり得ない(図24のスート3の欄参照)。しかし、基本ブロックD、E、Fの内部の命令間の依存関係が図26に示すようになっていた場合、すなわち、基本ブロックDにおける命令②と基本ブロックFにおける命令①とに依存関係があり、基本ブロックDにおける命令①と基本ブロックFとの間には依存関係がない場合、基本ブロックFの命令①は基本ブロックDの命令①と並列に実行することが可能である。したがって、この命令レベルでの依存関係を考慮することにより、スート3の最短処理時間を4と見積もることができる。

【0065】以上の処理を実現するため、第2の手法では、基本ブロック・コードスケジューラ21において、基本ブロック内部の命令の依存関係に関する情報を取得する。基本ブロック・コードスケジューラ21は、まず、命令間の依存関係を示す依存DAG(Directed Acyclic Graph)の全てのパスを求める。そして、得られたパスをその長さの降順に並べ替えておく。図27は、依存DAGの全てのパスを、ノードへのポインタを張ることによって保持するイメージを表す図である。

【0066】次に、基本ブロックに付された情報に基づいて、ハイパーブロック生成部22がPDGを作成し、

PDGをシリーズパラレルグラフに変換し、さらにシリーズパラレルグラフからシリーズパラレルネスト木を生成する行程は、第1の手法と同一である。したがって、ここでは詳細な説明を省略する。

【0067】次に、ハイパーブロック生成部22は、実行時間見積部23を用いて、シリーズパラレルネスト木における各ノードの実行時間を再帰的に見積もり、その結果に基づいてハイパーブロック選択処理を実行する(図10参照)。このハイパーブロック選択処理により、シリーズパラレルネスト木の各ノードに、独立したハイパーブロックとして扱うか否かを示す情報が付される。第2の手法におけるハイパーブロック選択処理は、基本的には第1の手法と同様の手順で実行されるが、基本ブロック内部の命令の依存関係を考慮し、必要に応じて当該基本ブロックのクリティカルパス長を変更する処理が実行される。以下、ハイパーブロック選択処理について、詳細に説明する。

【0068】ハイパーブロック選択処理の全体的な動作の流れは、図11を参照して説明した第1の手法と同様である。すなわち、まず、シリーズパラレルネスト木のノードの一つを処理対象とし(ステップ1101)、当該ノードの属性を調べ、パラレルスートか、シリーズスートか、基本ブロック(単一ノード)かを判断する(ステップ1102)。そして、判断結果に応じて、パラレルスートの実行時間見積処理(ステップ1103)、シリーズスートの実行時間を評価するための処理(ステップ1104、1105)、単一ノードの実行時間評価処理を行う(ステップ1106)。

【0069】また、パラレルスートの実行時間見積処理(ステップ1103)も、実行時間評価処理の内容を除き、全体的な動作の流れは図13を参照して説明した第1の手法と同様である。すなわち、まず、処理対象であるノードの子ノードに対して当該ハイパーブロック選択処理を再帰的に適用する(ステップ1301)。そして、パラレルスートの実行時間評価処理(ステップ1302)及び利得見積処理(ステップ1303)を行った後、得られた利得の値が0よりも大きいかなかを判断する(ステップ1304)。利得が0よりも大きい場合は、ノード選択処理を行う(ステップ1305)。利得が0以下である場合は、子ノード1と子ノード2の面積(すなわち、node.all: 当該ノードの延べ実行時間)の和を親ノード(すなわち、初めに処理対象としたノード)の面積とする(ステップ1306)。

【0070】図28は、第2の手法において、図13におけるステップ1302の実行時間評価処理の内容を説明するフローチャートである。また、図29は、図28の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図28を参照すると、まず、この処理の中でのみ有効な自由変数w1の値を1に初期化する(ステップ2801)。また、この処理の中でのみ有効な自

変数  $min$  及び  $min2$  の値を無限大に初期化する (ステップ2802)。そして、変数  $w1$  とハードウェアの並列度  $W$  とを比較し、同じかどうか調べる (ステップ2803)。変数  $w1$  が並列度  $W$  と同じでなければ、次に、この処理の中でのみ有効な自由変数  $w2$  を0に初期化し (ステップ2804)、この変数  $w2$  と変数  $w1$  とを比較して同じかどうか調べる (ステップ2805)。変数  $w1$  と変数  $w2$  とが同じでない場合、次に、子ノード1が並列度  $w2$  のときの最短処理時間と、子ノード2が並列度 ( $w1 - w2$ ) のときの最短処理時間とを比較し、大きい方の値を求める (ステップ2806)。そして、得られた値と、変数  $min$  の値とを比較し、小さい方の値を変数  $min$  の新たな値とする (ステップ2807)。また、子ノード1が並列度  $w2$  のクリティカルパス長を無視した場合の処理時間と、子ノード2が並列度 ( $w1 - w2$ ) のときのクリティカルパス長を無視した場合の処理時間とを比較し、大きい方の値を求める (ステップ2808)、そして、得られた値と、変数  $min2$  の値とを比較し、小さい方の値を新たな  $min2$  の値とする (ステップ2809)。この後、変数  $w2$  の値を1増加し (ステップ2810)、ステップ2805の比較に戻る。一方、変数  $w1$  と変数  $w2$  とが同じである場合は、現時点での変数  $min$  の値を親ノード (すなわち、初めに処理対象としたノード) が並列度  $w1$  のときの最短処理時間とし、現時点での変数  $min2$  の値を当該親ノードのクリティカルパス長を無視した場合の処理時間とする (ステップ2811)。そして、変数  $w1$  の値を1増加し (ステップ2812)、ステップ2802に戻る。そして、ステップ2803の比較において、変数  $w1$  と並列度  $W$  とが同じであれば、依存パス融合処理 (ステップ2813) を行った後に実行時間評価処理を終了する。

【0071】図30は、図28におけるステップ2813の依存パス融合処理の内容を説明するフローチャートである。なお、依存パスとは、依存DAGにおいて命令間の依存関係を表すパスである。また、図30において、 $node\_n\_exec\_pathl$  は、ノードにおける依存パスの数を示す。図30を参照すると、まず、親ノードにおける依存パスの数を子ノード1における依存パスの数と子ノード2における依存パスの数の和とする (ステップ3001)。次に、この処理の中でのみ有効な自由変数  $n1$ 、 $n2$ 、 $n$  を0に初期化する (ステップ3002)。そして、変数  $n$  と親ノードの依存パスの数とを比較し、同じかどうか調べる (ステップ3003)。変数  $n$  と親ノードの依存パスの数とが同じでない場合、次に、子ノード1の  $n1$  番目の依存パスの長さ、子ノード2の  $n2$  番目の依存パスとの長さを比較する (ステップ3004)。そして、子ノード1の  $n1$  番目の依存パスの方が長い場合は、親ノードの  $n$  番目の依存パスを、子ノード1の  $n1$  番目の依存パスとし、変数  $n1$  の値を1増加す

る (ステップ3005)。また、子ノード2の  $n2$  番目の依存パスの方が長い場合は、親ノードの  $n$  番目の依存パスを、子ノード2の  $n2$  番目の依存パスとし、変数  $n2$  の値を1増加する (ステップ3006)。ステップ3005またはステップ3006の後、変数  $n$  の値を1増加し (ステップ3007)、ステップ3003へ戻る。そして、変数  $n$  と親ノードの依存パスの数とが同じならば、依存パス融合処理を終了する (ステップ3003)。

【0072】以上の動作により、並列度  $W$  のときのパレールートの最短処理時間の見積もりが得られる。上記のように、第2の手法では、クリティカルパス長を無視した (すなわち、実行時間がクリティカルパス長を下回することを許す) 場合の処理時間の見積もり (図7の  $node\_best\_time2$ ) も計算される。この値は、後述するシリーズスートの実行時間の評価において用いられる。また、依存パス融合処理において、依存パスの融合とソートが行われる。子ノード1、2の依存パスは、基本ブロック・コードスケジューラ21により長いものから降順にソートしてあるので、これを用いて親ノードにおける依存パスが再構成される。

【0073】図31は、第2の手法において、図11におけるステップ1105のシリーズスートに対する実行時間評価処理の内容を説明するフローチャートである。また、図32は、図31の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図31を参照すると、まず、処理対象であるノードの延べ実行時間を、子ノード1、2の延べ実行時間の和とする (ステップ3101)。次に、クリティカルパス長再計算処理 (ステップ3102)、最短処理時間計算処理 (ステップ3103) を順次実行する。

【0074】図33は、図31におけるステップ3102のクリティカルパス長再計算処理内容を説明するフローチャートである。図33を参照すると、まず、この処理の中でのみ有効な自由変数  $n1$ 、 $idx$  を0に初期化する (ステップ3301)。そして、変数  $n1$  の値と子ノード1の依存パスの数とを比較し、同じかどうかを調べる (ステップ3302)。変数  $n1$  の値と子ノード1の依存パスの数とが同じでない場合、次に、この処理の中でのみ有効な自由変数  $n2$  を0に初期化する (ステップ3303)。そして、変数  $n2$  の値と子ノード2の依存パスの数とを比較し、同じかどうかを調べる (ステップ3304)。変数  $n2$  の値と子ノード2の依存パスの数とが同じでない場合、次に、子ノード1の  $n1$  番目の依存パスの最後のノードが、子ノード2の  $n2$  番目の依存パスの最初のノードに依存しているかどうか調べる (ステップ3305)。そして、依存しているならば、当該子ノード1の  $n1$  番目の依存パスと子ノード2の  $n2$  番目の依存パスとを結合し、親ノード (すなわち、初めに処理対象としたノード) の  $idx$  番目の依存パスと

する(ステップ3306)。ステップ3306の処理の後、及びステップ3305において、子ノード1の $n_1$ 番目の依存パスの最後のノードが、子ノード2の $n_2$ 番目の依存パスの最初のノードに依存していない場合、変数 $w_2$ の値を1増加し(ステップ3307)、ステップ3304に戻る。一方、ステップ3304において、変数 $n_2$ の値と子ノード2の依存パスの数とが同じならば、変数 $n_1$ の値を1増加し(ステップ3308)、ステップ3302へ戻る。そして、ステップ3302において、変数 $n_1$ の値と子ノード1の依存パスの数とが同じならば、処理を終了する。

【0075】以上のようにして、所定のシリーズストの子ノードにおいて、その子ノード中の命令の依存パスを連結し、この連結された依存パスの長さに基づいてソートした上で、当該シリーズストのクリティカルパス長を再計算する。これにより、子ノード中の命令の依存関係によっては、当該シリーズスト全体におけるクリティカルパス長を、子ノードのクリティカルパス長を単純に足した場合よりも短くすることができる。

【0076】図34は、図31におけるステップ3103の最短処理時間計算処理の内容を説明するフローチャートである。図34を参照すると、まず、親ノードの依存パスを長さの降順にソートする(ステップ3401)。次に、この処理の中でのみ有効な自由変数 $w$ を0に初期化する(ステップ3402)。そして、変数 $w$ とハードウェアの並列度 $W$ とを比較し、同じかどうかを調べる(ステップ3403)。変数 $w$ と並列度 $W$ とが同じでない場合、クリティカルパス長を考慮しない場合における親ノードの実行時間を、子ノード1、2のクリティカルパス長を考慮しない場合における実行時間の和とする(ステップ3404)。そして、変数 $w$ の値を1増加し(ステップ3405)、ステップ3403へ戻る。ステップ3403において、変数 $w$ と並列度 $W$ とが同じであれば、処理を終了する。

【0077】以上のようにして、子ノード1、2におけるクリティカルパス長を考慮しない場合における最短処理時間の和において、再計算された当該シリーズストのクリティカルパス長を下回らない値を当該シリーズストの最短処理時間と見積もることができる。

【0078】図35は、第2の手法において、図11におけるステップ1106の単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。また、図36は、図35の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。図35を参照すると、まず、この処理の中でのみ有効な自由変数 $w$ を1に初期化する(ステップ3501)。そして、変数 $w$ とハードウェアの並列度 $W$ とを比較し、同じかどうか調べる(ステップ3502)。変数 $w$ が並列度 $W$ と同じでなければ、次に、処理対象であるノードの最短処理時間を、基本ブロック中の延べ実行時間を変数 $w$ で割った値とす

る(ステップ3503)。また、処理対象であるノードのクリティカルパスを考慮しない場合における実行時間も、基本ブロック中の延べ実行時間を変数 $w$ で割った値とする(ステップ3504)。そして、ステップ3503で得られた値が基本ブロックのクリティカルパス長を下回ったかどうかを判断する(ステップ3505)。ステップ3503で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っている場合、当該ノードの最短処理時間を基本ブロックのクリティカルパス長とする(ステップ3506)。ステップ3503で算出された処理対象であるノードの最短処理時間が、基本ブロックのクリティカルパス長を下回っていない場合、またはステップ3506の終了後、変数 $w$ の値を1増加し(ステップ3507)、ステップ3502に戻る。そして、ステップ3502の比較において、変数 $w$ が並列度 $W$ と同じであれば、処理を終了する(ステップ3502)。

【0079】以上、説明した第2の手法は、基本ブロック・コードスケジューラ21において依存DAGのパス(依存パス)のソートを行う。この処理は、依存パスの数を $m$ とした場合、 $m \times \log m$ に比例する計算時間を要する。また、依存パスの連結には最長で $m^2$ に比例する計算時間を要し、連結された依存パスのソートにやはり $m \times \log m$ に比例する計算時間を要する。したがって、上述した処理は、最長で $n \times m^2$ に比例する計算時間を要することとなる。しかしながら、プログラム中の最適化しようとする部分が多く、基本ブロックに分割されている場合、依存パスの数 $m$ は小さな値となる。したがって、実践的には大きな計算時間を要しないで実行が可能である。

【0080】次に、第2の手法による最短処理時間の計算例を説明する。図37は、図2、3のプログラムに対して第2の手法により得られた最短処理時間の見積もりを、図6のシリーズパラレルネスト木における各ノード(スト)に対して、並列度1～6の各場合で求めた結果を示す図である。また、図38は、基本ブロックのクリティカルパス長を無視した場合における最短処理時間の見積もりを、同様にして求めた結果を示す図である。ここで、基本ブロックD、E、Fの内部における命令の依存関係は、図26に示したようになっている。

【0081】したがって、第1の手段により得られた最短処理時間の見積もりを示す図24と、図37とを比較すると、基本ブロックD、E、Fの命令の依存関係が、スト3における見積もりの結果に現れている。スト3は、基本ブロックFとスト4とで構成されたシリーズストである。ここで、スト3における並列度6の場合の最短処理時間は図37によれば4であり、図24の場合における5よりも1だけ少なくなっている。そして、この値は、図38における該当個所の最短処理時間が4であることから、再計算されたクリティカルパス長

を下回らないことがわかる。したがって、この場合のスト3の最短処理時間は4と見積もられ、第1の手法の場合と比べて最適化が進んでいる。

【0082】

【発明の効果】以上説明したように、本発明によれば、プログラムの所定の領域に対して高効率かつ適切なハイパーブロックの生成を行うことができるため、最適化処理において、ある程度実行可能性が高い多くのパスにおける実行効率を向上させることができる。

【図面の簡単な説明】

【図1】 本発明の実施の形態におけるコンパイラの構成を説明する図である。

【図2】 処理対象であるプログラムの最適化処理を行う領域の制御フローグラフと当該部分の命令列のリストを示す図である。

【図3】 図2に示したプログラム領域において、node.clを縦の長さとし、node.allを面積とする矩形領域で基本ブロックを表した図である。

【図4】 図3から作成されたPDGを示す図である。

【図5】 図4のPDGから変換されたシリーズパラレルグラフを示す図である。

【図6】 図5のシリーズパラレルグラフから生成されたシリーズパラレルネスト木を示す図である。

【図7】 本実施の形態の動作説明に用いる記号を定義した図表である。

【図8】 PDGをシリーズパラレルグラフに変換するアルゴリズムを示す疑似プログラムを示す図である。

【図9】 シリーズパラレルグラフからシリーズパラレルネスト木を生成するアルゴリズムを示す疑似プログラムを示す図である。

【図10】 本実施の形態におけるハイパーブロック生成部の全体動作を説明するフローチャートである。

【図11】 ハイパーブロック選択処理の全体的な動作の流れを示すフローチャートである。

【図12】 図11に対応する動作のアルゴリズムを示す疑似プログラムを示す図である。

【図13】 パラレルストに対する実行時間見積処理の内容を説明するフローチャートである。

【図14】 実行時間評価処理の内容を説明するフローチャートである。

【図15】 図14の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図16】 利得見積処理の内容を説明するフローチャートである。

【図17】 図16の利得見積処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図18】 ノード選択処理の内容を説明するフローチャートである。

【図19】 図18のノード選択処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図20】 シリーズストに対する実行時間評価処理の内容を説明するフローチャートである。

【図21】 図20の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図22】 単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。

【図23】 図22の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図24】 最短処理時間の見積もりを、図6のシリーズパラレルネスト木における各ノード(スト)に対して、並列度1~6の各場合で求めた結果を示す図である。

【図25】 最短処理時間の見積もりを、図6のシリーズパラレルネスト木における各ノード(スト)に対して、並列度1~3の各場合で求めた結果を示す図である。

【図26】 図3に示した基本ブロックC、D、E、Fの内部の命令における依存関係を説明する図である。

【図27】 依存DAGの全てのパスを、ノードへのポインタを張ることによって保持するイメージを表す図である。

【図28】 第2の手法において、実行時間評価処理の内容を説明するフローチャートである。

【図29】 図28の実行時間評価処理の動作アルゴリズムを示す疑似プログラムを示す図である。

【図30】 依存パス融合処理の内容を説明するフローチャートを示す図である。

【図31】 第2の手法において、シリーズストに対する実行時間評価処理の内容を説明するフローチャートである。

【図32】 図31の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。

【図33】 クリティカルパス長再計算処理内容を説明するフローチャートである。

【図34】 最短処理時間計算処理の内容を説明するフローチャートである。

【図35】 第2の手法において、単一ノードに対する実行時間評価処理の内容を説明するフローチャートである。

【図36】 図35の実行時間評価処理の動作アルゴリズムを示す疑似プログラムである。

【図37】 第2の手法により得られた最短処理時間の見積もりを、図6のシリーズパラレルネスト木における各ノード(スト)に対して、並列度1~6の各場合で求めた結果を示す図である。

【図38】 基本ブロックのクリティカルパス長を無視した場合における最短処理時間の見積もりを、同様にして求めた結果を示す図である。

【図39】 処理対象のプログラムの構成を基本ブロックで表現した例を示す図である。



(17)

特開2002-116916

【図40】 パラレルスートの実行時間を構成要素のパラメータ (d、w) で表現できることを説明する図である。

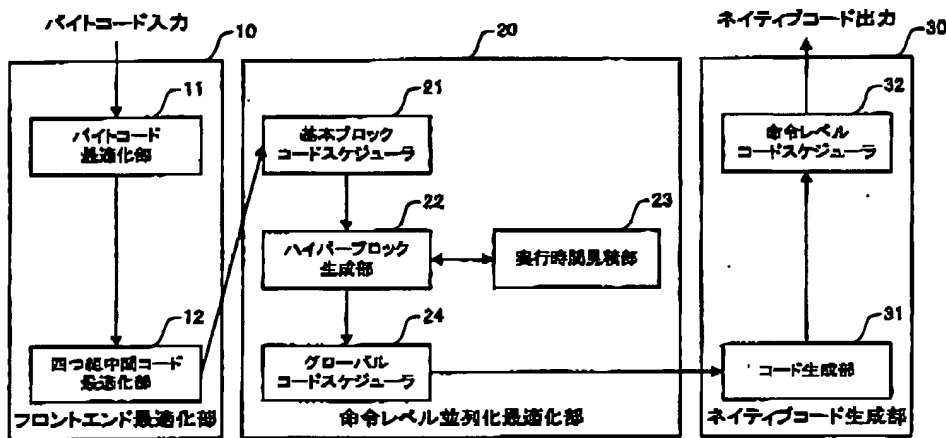
【図41】 モデル化された基本ブロックによるパラレルスートの例を示す図である。

【図42】 基本ブロック内部における疎間を説明する図である。

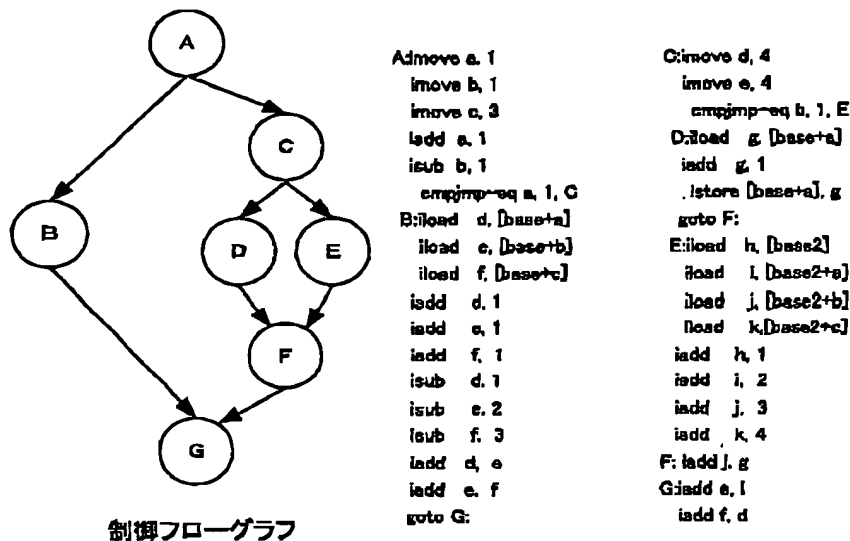
【符号の説明】

10…フロントエンド最適化部、11…バイトコード最適化部、12…四つ組中間コード最適化部、20…命令レベル並列化最適化部、21…基本ブロック・コードスケジューラ、22…ハイパーブロック生成部、23…実行時間見積部、24…グローバル・コードスケジューラ、30…ネイティブコード生成部、31…コード生成部、32…命令レベル・コードスケジューラ

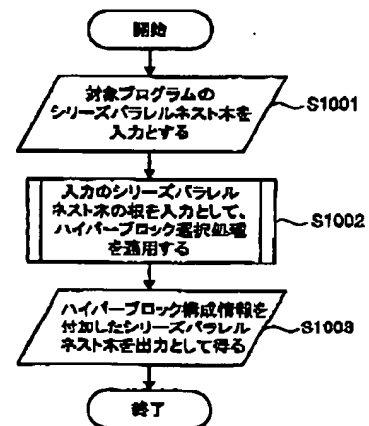
【図1】



【図2】



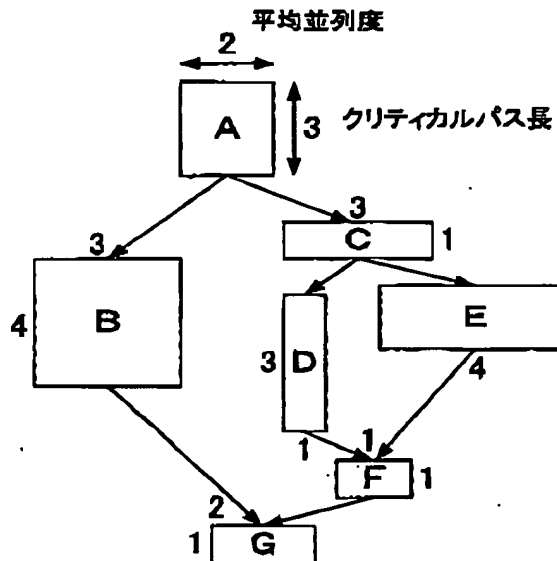
【図10】



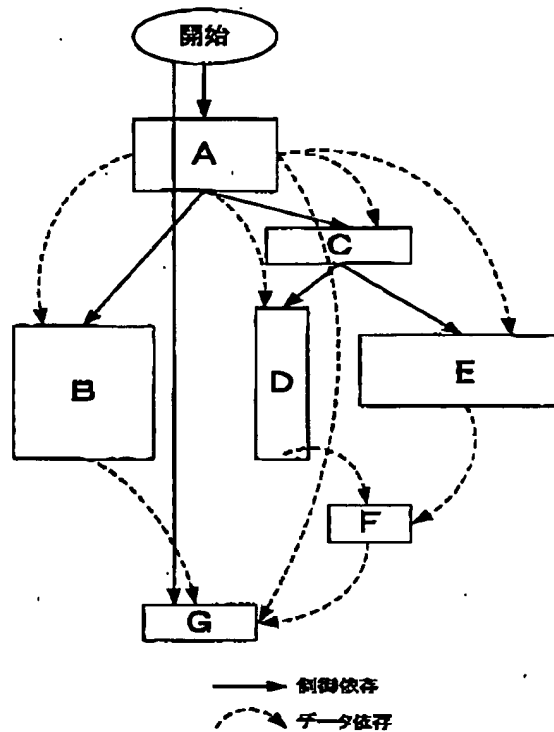
(18)

特開2002-116916

【図3】



【図4】



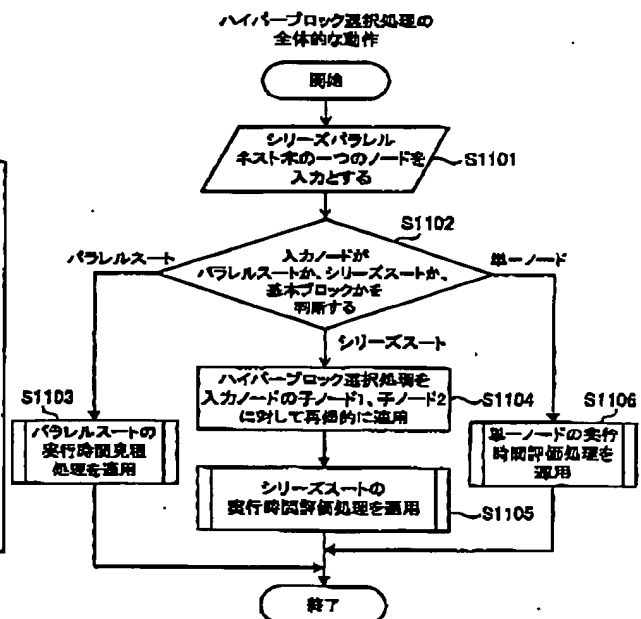
【図8】

PDGをシリーズパラレルグラフに変換する疑似プログラム

```

Input: PDG, dominator tree of the PDG
Output: transformed PDG
for n in all the PDG node
begin
  if (the number of predecessor of n is more than 1)
  begin
    for all the combination (p1, p2) of the predecessor of n
    begin
      if (p1 dominates p2)
        remove an edge from p1 to n
      if (p2 dominates p1)
        remove an edge from p2 to n
    end
  end
end
end
    
```

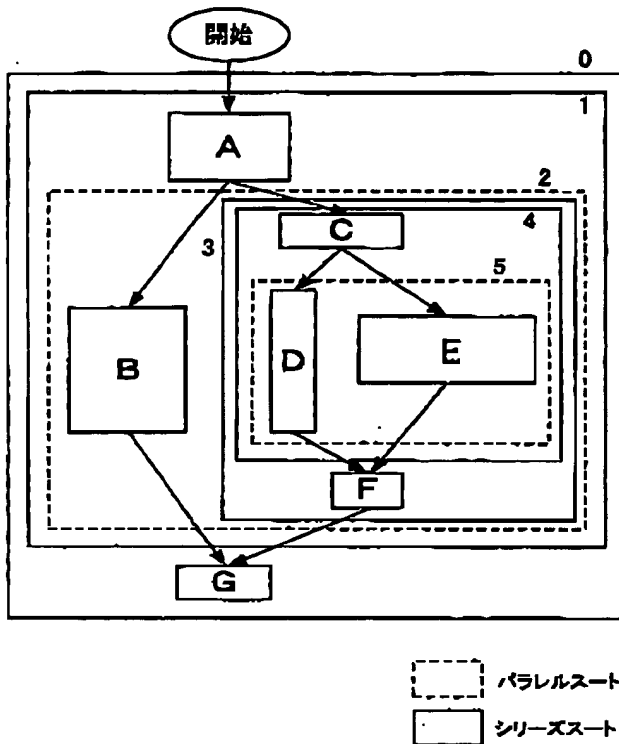
【図11】



(19)

特開2002-116916

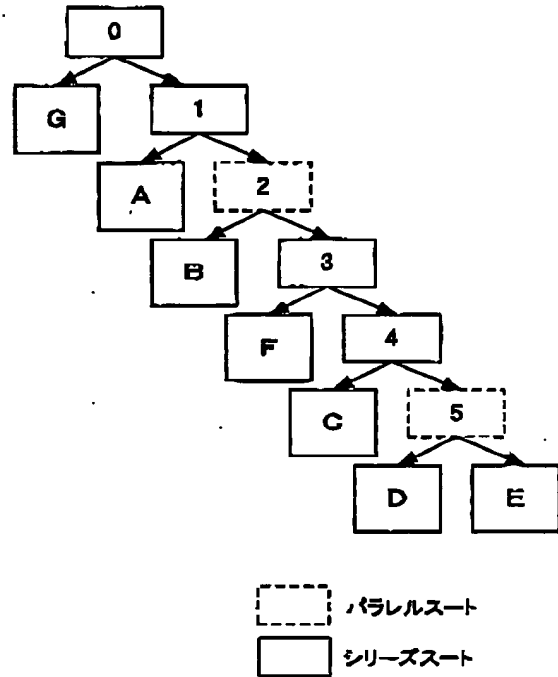
【図5】



【図7】

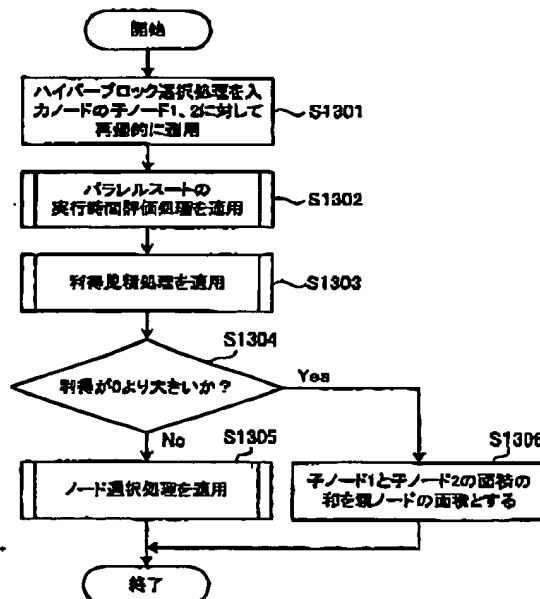
W	ハードウェアの並列度(定数)。
w, w1, w2	主にハードウェアの並列度を表す。ループの回数を表すとして使われる。それぞれの出現で局所的。
node	シリーズ/パラレルリスト木の一つのノードを表す。
node.attributes	ノードが基本ブロック、シリーズスト、パラレルストのいずれであるかを表す。
node.cil	ノードが表すストのクリティカルパス長(ノードが基本ブロックである場合にのみ定義される)。
node.a3	ノードの遅延実行時間。
node.child1 node.child2	ノードのリストの内側のシリーズスト、パラレルスト、基本ブロックを表す(ノードがシリーズストまたはパラレルストである場合にのみ定義される)。
node.dep_path[n] node.dep_path_in node.dep_path_out node.dep_path_length node_in_dep_path	ノードが表す領域内の依存パスのリスト。ハイパーブロック選択処理の第2の手法で、基本ブロックのコードスケジューリングを行う際に生成する。dep_path_lengthは、依存パスのヘッド位置にあるDAGノードへのポイント。dep_path_outは、依存パスのテール位置にあるDAGノードへのポイント。
node.best_time[w]	ノードが表すストの最長処理時間の見積もり。定数値は $1 \leq w \leq W$ 。
node.best_time2[w]	クリティカルパス長を無視した場合のノードの処理時間の見積もり。実行時間がクリティカルパス長以下になる場合を許す。ハイパーブロック選択処理の第2の手法で見積もられ、シリーズストの実行時間の見積もりに用いられる。定数値は $1 \leq w \leq W$ 。
node.hyper_block	値としてTRUEかFALSEかをとる。独立のハイパーブロックを生成する場合、TRUEになる。

【図6】



【図13】

パラレルストの実行時間見積処理



(20)

特開2002-116916

【図9】

シリーズ/パラレルネスト木を生成する擬似プログラム

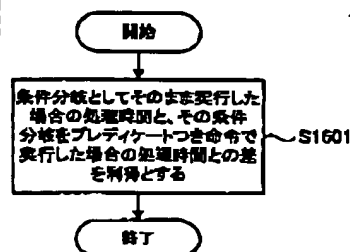
```

Input: Series-Parallel Graph
Output: suite nest tree

apply following "algorithm" to the header node of Series-Parallel Graph
algorithm(node)
begin
  allocate new root R of suite nest tree
  if (node has more than 1 successor)
  begin
    for each successor s(i) of node
    begin
      put (recursively called) algorithm(s(i)) as the leaf of R
    end
    R.attribute = parallel suite;
    return R;
  end
  else
  begin
    put (recursively called) algorithm(the successor of node) as the leaf of R
    R.attribute = series suite;
    return R;
  end
end
end
  
```

【図16】

利得見積処理



【図12】

ハイパーブロック選択処理の擬似プログラム

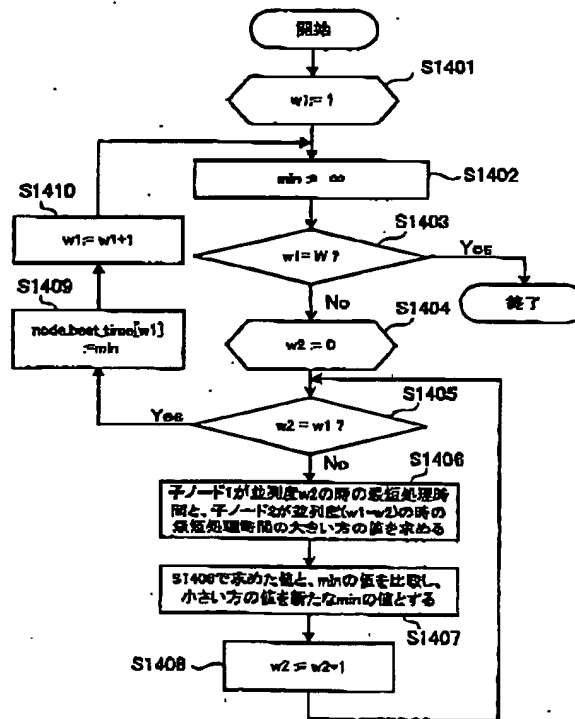
```

Hyper_Block_Selector(Node node)
begin
  switch(node.attribute)
  begin
    case: single node
    single node
    single_node_evaluator(node);
    return;
  break;
  case: Series-Suite
  Series Suite
  Hyper_Block_Selector(node.child1);
  Hyper_Block_Selector(node.child2);
  series_suite_evaluator(node.child1, node.child2);
  break;
  case: Parallel-Suite
  Parallel Suite
  Hyper_Block_Selector(node.child1);
  Hyper_Block_Selector(node.child2);
  // do parallel schedule
  parallel_suite_evaluator(node.child1, node.child2);

  // whether hyperblock is formed or not
  gain = gain_estimator(node.child1, node.child2, node);
  if (gain <= 0)
  begin
    //chooses one with some heuristics
    //the other is flagged as separate hyper_block
    node_selector(node.child1, node.child2);
  end
  else
    node.all = node.child1.all + node.child2.all;
  break;
  end of switch
end
end
  
```

【図14】

パラレルスイートの実行時間評価処理



【図15】

パラレルスイートの実行時間評価処理の擬似プログラム

```

procedure parallel_suite_evaluator(node, child1, child2)
begin
  int min, w1, w2;
  for w1=1 to W
  begin
    min = infinite;
    for w2=1 to w1
    begin
      min = min( min, max(child1.best_time[w2], child2.best_time[w1-w2] );
    end
    node.best_time[w1] = min;
  end
end
end

```

【図17】

利得見積処理の擬似プログラム

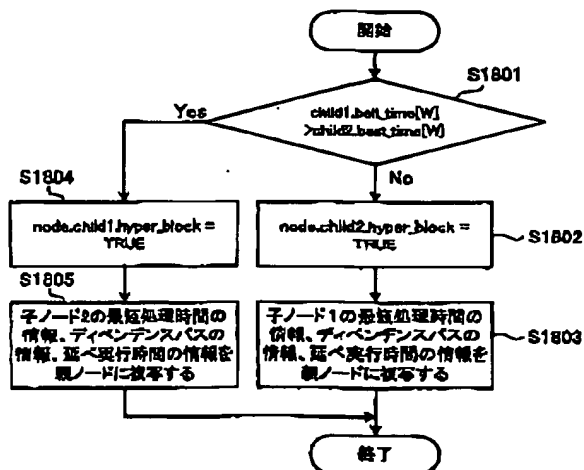
```

procedure gain_estimator(node)
  return branch_penalty+(node.child1.best_time[W]+node.child2.best_time[W])/2
  - node.best_time[W];

```

【図18】

ノード選択処理



【図19】

ノード選択処理の擬似プログラム

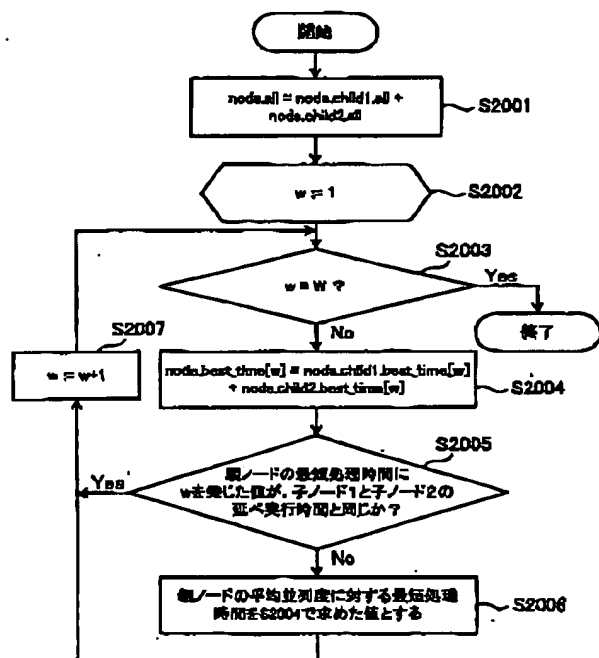
```

procedure node_selector(child1, child2)
begin
  if(child1.best_time[W] > child2.best_time[W])
  begin
    node.child2.hyper_block=TRUE;
    copy node.child1.best_time to node.best_time;
    copy node.child1.dep_path to node.dep_path;
  end
  else
  begin
    node.child1.hyper_block=TRUE;
    copy node.child2.best_time to node.best_time;
    copy node.child2.dep_path to node.dep_path;
  end
end
end

```

【図 20】

シリーズスイートの実行時間評価処理



【図 21】

シリーズスイートの実行時間評価処理の擬似プログラム

```

procedure series_suite_evaluator(node, child1, child2)
begin
  int w;
  node.all = node.child1.all + node.child2.all;
  for w=1 to W
  begin
    node.best_time[w] = node.child1.best_time[w] + node.child2.best_time[w];
    if (node.best_time[w] * w != node.child1.all + node.child2.all)
      node.best_time[
        (node.child1.all + node.child2.all) / node.best_time[w]
      ] = node.best_time[w];
  end
end
  
```

【図 23】

単一ノードの実行時間評価処理の擬似プログラム

```

procedure singlenode_evaluator(node, w)
return max(node.all/w, node.cl);
  
```

【図 36】

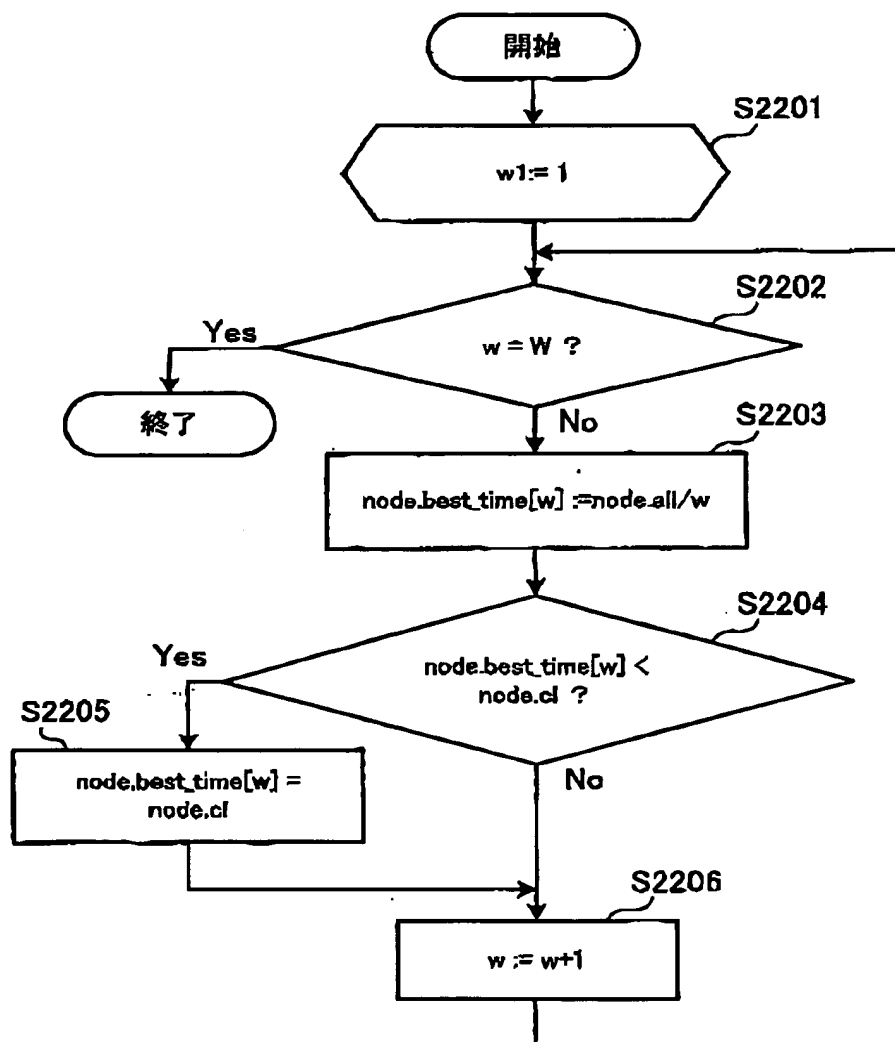
単一ノードの実行時間評価処理の擬似プログラム

```

procedure singlenode_evaluator(node)
begin
  int w;
  for w = 1 to W
  begin
    node.best_time2[w] = node.all/w;
    node.best_time[w] = max(node.all/w, node.cl);
  end
end
  
```

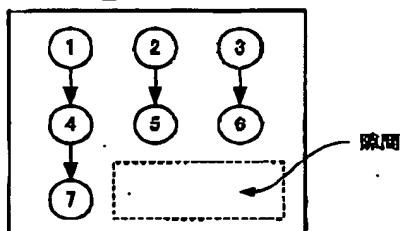
【図22】

単一ノードの実行時間評価処理



【図42】

基本ブロック



(24)

特開2002-116916

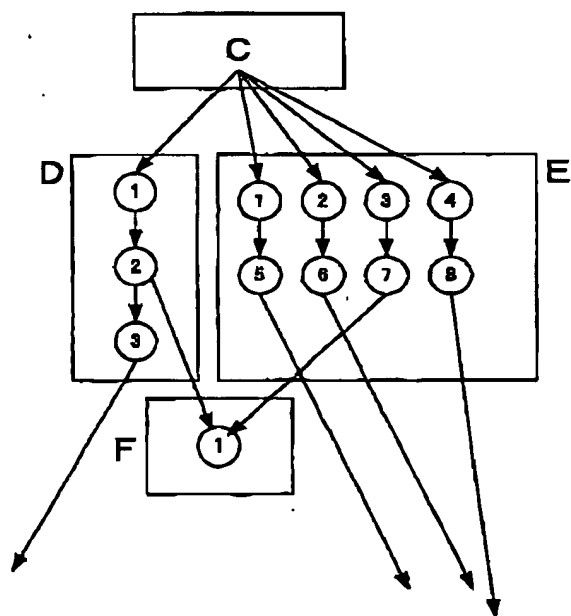
【図24】

スタート/ 並列度	1	2	3	4	5	6
A	6	3	3	3	3	3
B	12	6	4	4	4	4
C	3	2	1	1	1	1
D	3	3	3	3	3	3
E	8	4	3	2	2	2
F	1	1	1	1	1	1
G	2	1	1	1	1	1
O(*)	35	19	16	15	10	10
1	33	18	15	14	9	9
2	27	15	12	11	6	6
3	15	11	6	6	5	5
4	14	10	5	4	4	4
5	11	8	4	3	3	3

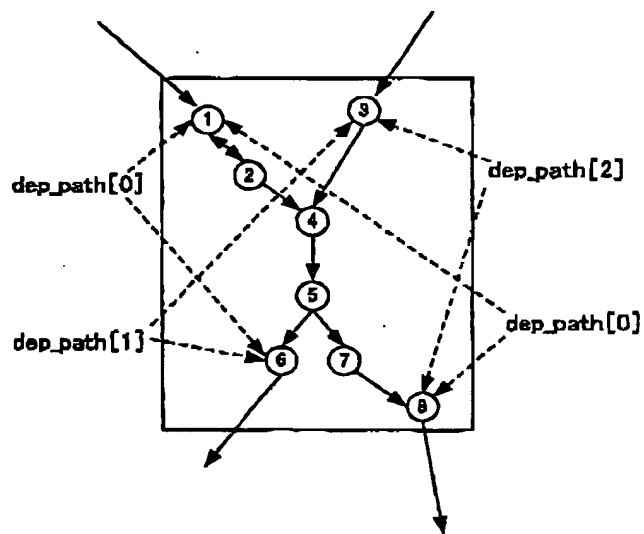
【図25】

スタート/ 並列度	1	2	3
A	6	3	3
B	12	6	4
C	3	2	1
D	3	3	3
E	8	4	3
F	1	1	1
G	2	1	1
O(*)	20	10	8
1	18	8	7
2	12	6	4
3(*)	16	11	6
4	15	10	5
5	12	8	4

【図26】



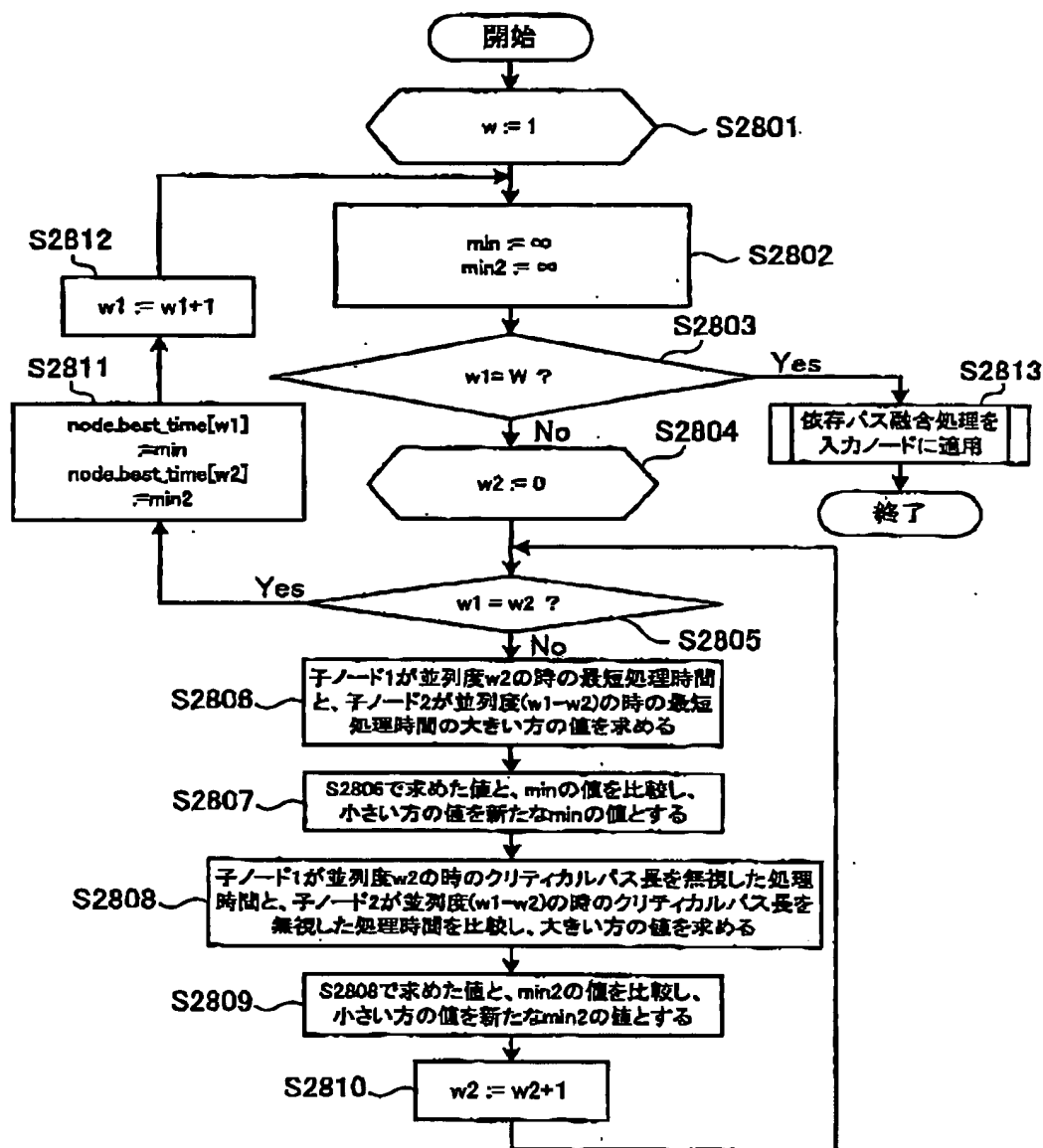
【図27】





【図28】

パラレルスート実行時間評価処理



【図29】

パレルスートの実行時間評価処理の類似プログラム

```

procedure parallel_suta_evaluation(node, child1, child2)
begin
  int min = infinity;
  int min2 = infinity;
  for w1 = 1 to W, n1, n2;
  for w2 = 1 to W
  begin
    min = min(min, max(child1.best_time[w2], child2.best_time[w1-w2]));
    min2 = min(min2, max(child1.best_time[w2], child2.best_time[w1-w2]));
  end
  node.best_time[w1] = min;
  node.best_time[w2] = min2;
end

// merge child1.dep_path & child2.dep_path into node.dep_path
n1 = 0; n2 = 0;
for n = 0 to node.n_dep_path - 1
  if (child1.dep_path[n].length > child2.dep_path[n2].length)
    node.dep_path[n] = child1.dep_path[n];
  else
    node.dep_path[n] = child2.dep_path[n2];
  end
end
node.n_dep_path = child1.n_dep_path + child2.n_dep_path;
end

```

【図32】

シリーズートの実行時間評価処理の類似プログラム

```

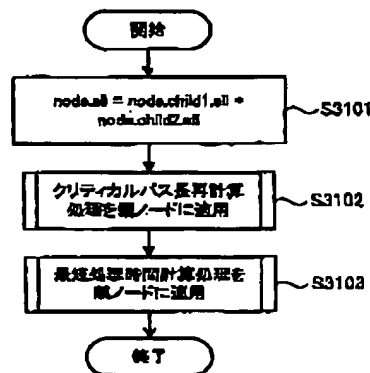
procedure series_suta_evaluation(child1, child2, w)
begin
  int w, n1, n2, l;
  //critical path calculation
  for n1 = 0 to child1.n_dep_path - 1
  begin
    for n2 = 0 to child2.n_dep_path - 1
    begin
      if (child1.dep_path[n1].out depends on child2.dep_path[n2].in)
      begin
        concat(child1.dep_path[n1] & child2.dep_path[n2]) as
        node.dep_path[n];
        node.dep_path[n].length =
        child1.dep_path[n1].length + child2.dep_path[n2].length;
      end
    end
  end
  sort node.dep_path[n];
  node.cl = node.dep_path[n];

  //best_time calculation
  for w = 1 to W
  begin
    tmp = child1.best_time[w] + child2.best_time[w];
    node.best_time[w] = max(node.cl, tmp);
    node.best_time[w] = tmp;
  end
end of procedure

```

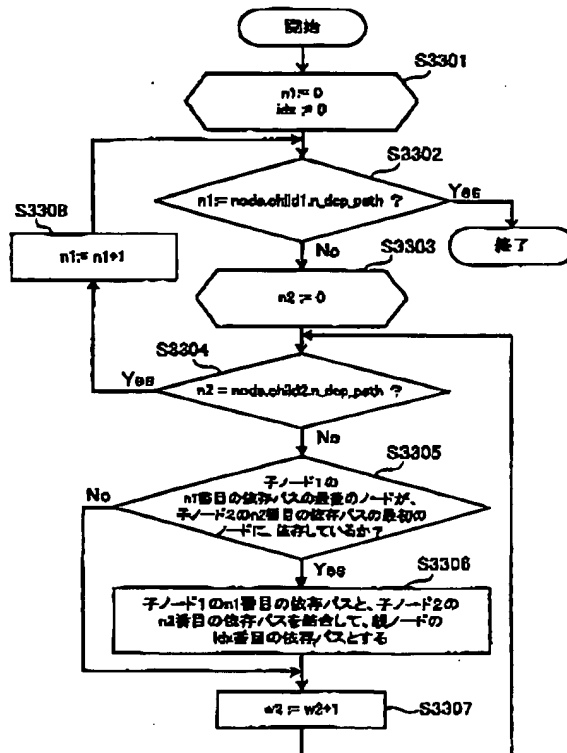
【図31】

シリーズートの実行時間評価処理



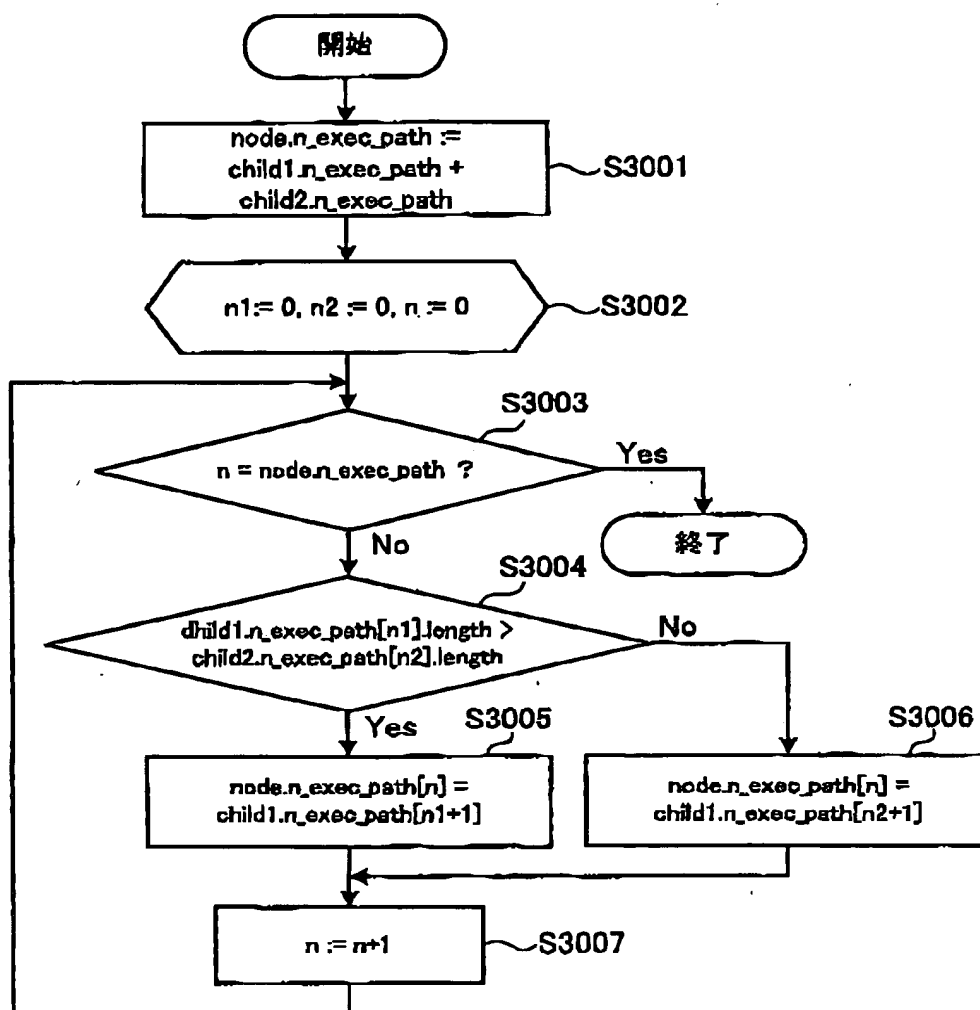
【図33】

クリティカルパス長再計算処理

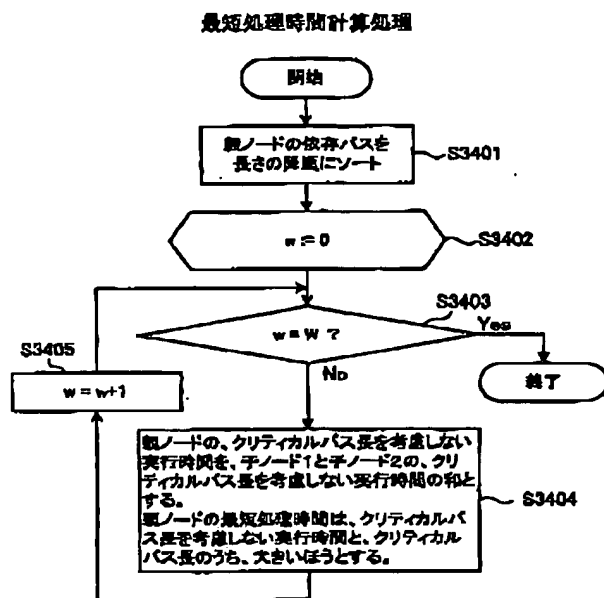


【図30】

依存パス融合処理



【図34】

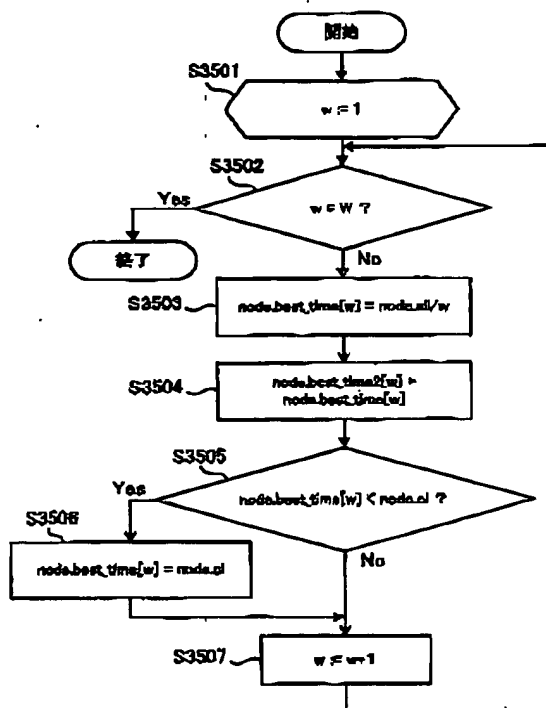


【図37】

スタート/ 並列度	1	2	3	4	5	6
A	6	3	3	3	3	2
B	12	6	4	4	4	4
C	3	2	1	1	1	1
D	3	3	3	3	3	3
E	8	4	3	2	2	2
F	1	1	1	1	1	1
G	2	1	1	1	1	1
0	36	20	16	13	10	10
1	34	19	15	14	9	9
2	28	16	12	11	6	6
3	16	11	8	6	5	4
4	15	10	5	4	4	4
5	12	8	4	3	3	3

【図35】

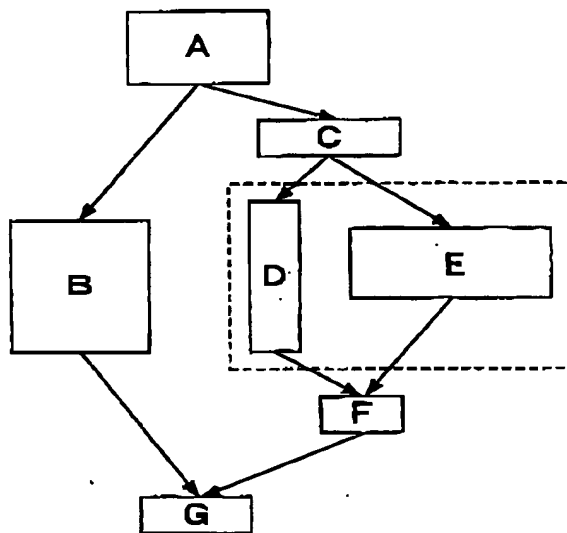
単一ノードの実行時間評価処理



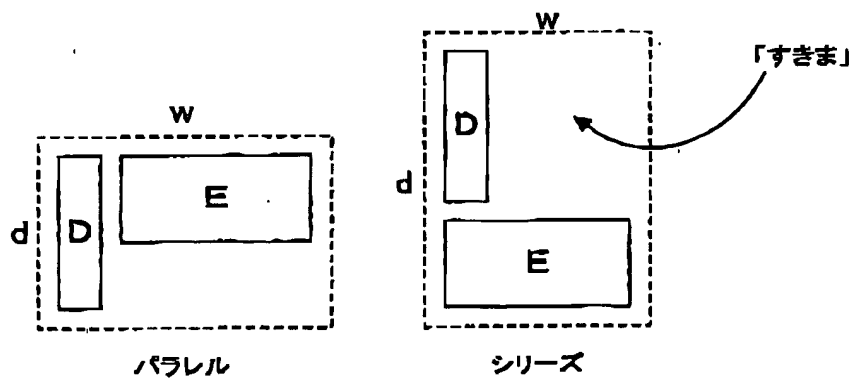
【図38】

スタート/ 並列度	1	2	3	4	5	6
A	6	3	2	2	1	1
B	12	6	4	3	2	2
C	3	2	1	1	1	1
D	3	2	1	1	1	1
E	8	4	3	2	2	1
F	1	1	1	1	1	1
G	2	1	1	1	1	1
0	36	20	15	14	8	8
1	34	19	14	13	7	7
2	28	16	12	11	6	6
3	16	11	8	6	5	4
4	15	10	5	5	4	3
5	12	8	4	4	3	2

【図39】



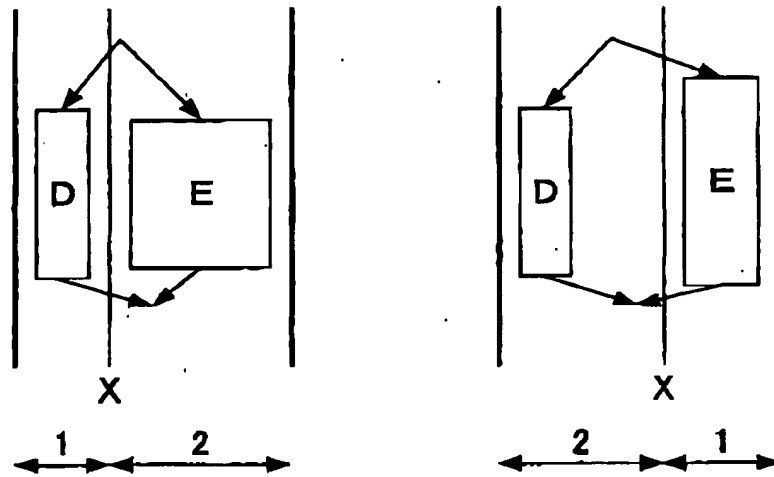
【図40】



(30)

特開2002-116916

【図41】



フロントページの続き

(72)発明者 田端 邦男  
神奈川県大和市下鶴間1623番地14 日本ア  
イ・ビー・エム株式会社 東京基礎研究所  
内

(72)発明者 小松 秀昭  
神奈川県大和市下鶴間1623番地14 日本ア  
イ・ビー・エム株式会社 東京基礎研究所  
内

Fターム(参考) 5B013 DD04  
5B033 CA22  
5B081 CG24 CC32